Indiana University-Purdue University Fort Wayne
Department of Civil and Mechanical Engineering
Department of Electrical and Computer Engineering
ECE 410 – ENGR 411
ECE 405 – ECE 406
Capstone Senior Design Project
Report #2

Project Title: Intraoperative Red Blood Cell Warmer

Team Members:
- Luis Bertran (ME)
- Austin Gagnon (EE)
- Brandon Hill (ME)
- Bradley Sordelet (ME)
- Jon Wesner (CPE)
- Peter Wirges (CPE/EE)

Faculty Advisors:
- Dr. Hosni Abu-Mulaweh (ME)
- Dr. Chao Chen (ECE)
- Dr. Bongsu Kang (ME)

Date: December 5, 2016
Table of Contents

Acknowledgments .......................................................................................................................... 3

Abstract/Summary .......................................................................................................................... 4

Section I: Building ............................................................................................................................ 7
  Build Stages .................................................................................................................................. 8
  Initial Steps to Start the Build ....................................................................................................... 8
  Machining of the Agitation System and Heating System Parts .................................................... 8
  Assembly of the Agitation System and Heating System ............................................................... 11
  Assembly of the Main Structure and Installation of Sub-Systems ................................................ 13
  Installation of the Electrical Components .................................................................................... 21
  Installation of Temperature Sensors ............................................................................................. 23
  Installation of the power supply and power switch ...................................................................... 25
  Connection of sensors to microcontroller .................................................................................... 26
  Connection of Raspberry Pi 3 to microcontroller ......................................................................... 27
  Wiring of Motor and Fan .................................................................................................................. 28
  Hardware Changes ......................................................................................................................... 30
  Design of Microcontroller PCB .................................................................................................... 35
  Design of Switching PCB .............................................................................................................. 38
  Design/Implementation of Software ............................................................................................. 41
  Build Challenges and Changes ..................................................................................................... 48

Section II: Testing and Evaluation ............................................................................................... 51
  Requirements and Specifications .................................................................................................. 52
  Door Switch ................................................................................................................................ 52
  Heating System Verification ......................................................................................................... 52
  Agitation ...................................................................................................................................... 59
  External Case Temperature ......................................................................................................... 59
  Weight and Stability ...................................................................................................................... 60
  Heat Loss .................................................................................................................................... 60
  Testing Challenges and Changes ................................................................................................. 63

Section III: Cost/Budget ............................................................................................................... 71
  Conclusion .................................................................................................................................. 73
Acknowledgments
We would like to thank Zimmer Biomet for their sponsorship of this project, and Dr. Chao Chen, Dr. Hosni Abu-Mulaweh, and Dr. Bongsu Kang for being our advisors on this project.
Abstract/Summary
This senior design project was sponsored by Zimmer Biomet, a biomedical company located in Warsaw, Indiana. The company owns the patent on a chemical called Rejuvesol, and is working on developing an accompanying system for it. Rejuvesol is capable of restoring the ability of red blood cells (RBC) to transport oxygen after they had been stored for more than two weeks. For this solution to work, among other things, the RBC have to be incubated at a temperature between 37°C and 41°C for an hour while being agitated. The focus of this project was to design and implement an Intraoperative Red Blood Cell Warmer (IRBCW) capable of performing these functions as specified by Zimmer.

The design process for this project can be summarized as follows: A problem statement was defined, which listed all of the requirements expected of the system. The system was then broken down into several sub-systems, and several concepts were developed for each of them. Attributes were selected for each sub-system and given weights. Each teammate individually ranked each concept on these attributes from best to worst. Using the attribute weights and averages of all of the members, the best concept was selected. After determining each subsystem, the selected concepts were combined to create the final design of the system. Lastly, a cost analysis of the design system was performed to determine the potential cost of the system.

The IRBCW was designed to meet all of the requirements listed below. During the second semester, the IRBCW was constructed and tested to verify that it met these requirements. These requirements were:

- The unit should be able to raise the temperature of two RBC units simultaneously from 38°C ±2°C to 37-41°C ±1°C in 10-15 minutes.
- The system should be able to maintain the desired temperature for at least one hour
- The RBC units should be constantly agitated during the cycle at a rate of 10rpm.
- The software should be user friendly and let the user input a desired temperature, display the desired temperature, the current temperature, and the cycle time.
- Other requirements are that the final device should employ some form of data logging for temperature and time, that it must be IV pole mountable, and that the total cost is less than $5,000.

The necessary parts and components were ordered and delivered successfully. The polycarbonate housing, heat duct, and agitation basket were sent out to be machined and then shipped to the design team. Once all of the parts and components were received, construction of the IRBCW started with assembling the polycarbonate plates, machining the agitation components, and installing the inner cavity with the agitation basket and motor attached to it. After the inner cavity was completed, the outer housing was constructed with the remaining polycarbonate pieces. Electrical components were designed, modified, machined, tested, and installed. Finally, the top was installed with the lid assembly and the IV pole clamps were attached as well. The entire build process of the IRBCW will be detailed later in the report.

When the IRBCW was fully constructed, it was tested to verify that all of the requirements listed above were achieved. From these tests, the IRBCW was capable of raising the external temperature of a RBC
from a worst case of 37°C to between 37-41°C±0.5°C in just over ten minutes, and maintaining that temperature for at least an hour. Despite this, additional tests showed the actual, internal bag temperature did not respond as desired. This temperature did reach and maintain the desired setpoint, however, it took about 4 times longer than the external temperature, and did not track on the display during this time. This issue is detailed later in the report, but it is important to note here that it cannot be rectified without significant modification to the design. The agitation basket was capable of constantly agitating two RBCs at 10 RPM throughout this entire process. The IRBCW features an attractive user interface while the user can set a temperature and see the current temperature and cycle time. Lastly, the system is capable of being mounted on an IV pole and cost $4,359.69 to complete which is under the set $5,000 budget.
Section I: Building
The material in this section will lay out the steps that were taken to build the IRBCW unit. Several steps were involved to start the build, including initial parts ordering and quoting. With the initial steps underway, the focus then shifted to the mechanical machining, modifications, and assembly of the IRBCW unit. In addition to the mechanical aspects, many electrical steps were taken. Details of the electrical component installation, wiring, microcontroller design, switching design, and software design are also described in detail. Challenges and changes to both the mechanical and electrical portions of this project are detailed in the later portion of this section.

**Build Stages**

Several steps were taken to build the RBC unit. A brief list outlines the general stages taken to build the unit.

- Send out list of all purchased parts to be ordered
- Compile polycarbonate part prints and send out for quotes
- Compile stainless steel sheet metal part prints and send out for quotes
- Compile aluminum part prints for in house machining
- Receive sheet metal and polycarbonate machining quotes and coordinate with Zimmer/Biomet to proceed with order
- Receive raw aluminum stock
- Machine all aluminum parts for agitation and heating system
- Receive polycarbonate structure pieces and assemble main structure
- Assemble agitation system
- Assemble heating system
- Install heating and agitation system into main structure
- Install all electrical components and run wiring

**Initial Steps to Start the Build**

The first step in building the RBC unit was to collect all part information. This included both purchased parts and custom machined parts. A purchased part list with all quantities and suppliers was developed earlier in the design stage of this project. This list was passed on to Zimmer/Biomet for purchasing. Next, the custom machined parts had to be addressed. The machined parts were broken down into three categories including sheet metal, polycarbonate, and aluminum pieces. With the capabilities and machinery available to the design team, it was decided to machine all of the aluminum pieces in house by the design team. The polycarbonate and sheet metal work were to be done by specialized shops. The part prints for these were sent out to multiple suppliers for quote. Once the quotes came back and decided upon, the parts were ordered and to be received in time for the build. With all the parts accounted for and on the way, the direction of the build shifted to the machining of the aluminum parts.

**Machining of the Agitation System and Heating System Parts**

In order to help keep the cost of the RBC unit down, it was decided to order aluminum stock and machine the pieces in-house. All of the aluminum pieces were made on a manual lathe and a manual mill with digital readout. First, the pieces were cut rough to size with a band saw. Then, these pieces were deburred and prepped to be machined. As seen in Figure 1, the parts have gone through several
processes to get to this point. They have been rough cut, cut to size, bored, drilled, countersunk, and chamfered. Figure 2 is another view of one of the parts getting chamfered with an endmill.

Figure 1: Machining of Aluminum Parts on Mill
From here, the parts need to be deburred as seen in Figure 3. This buffing wheel lightly sands all of the sharp machined edges in order to make the part look better and safer to handle. Once all of the parts were cleaned up, the last step of machining began. Tapping of the threaded holes can be seen in Figure 4. A hand tap and tapping lubricant were used to make all of the threaded holes. This left chips and oil inside all of the parts so a final cleaning was done to prepare the parts for assembly.
Assembly of the Agitation System and Heating System

Now that the parts have been machined, they needed to be assembled to make sure that everything worked as designed. As seen in Figure 5, the pivot and drive arms of the agitation system were bolted together using purchased bushings, screws, and shoulder bolts. The DC gear motor and bag basket were also installed to the agitation system, as seen in Figure 6, and ready for installation into the main structure.
Next, the aluminum mounting pieces were installed to the sheet metal duct work of the heating system. As seen in Figure 7, the aluminum pieces were inserted and bolted to the duct work. A thin layer of high temperature silicone was used to seal these parts and stop any air leakage during use. These aluminum inserts allowed the team to mount the fan and the duct work to the main structure of the build.
Assembly of the Main Structure and Installation of Sub-Systems

The first step of assembling the polycarbonate was to build the internal heat chamber where the RBC bags will be located. The six polycarbonate pieces were joined together with screws to create the chamber shown in Figure 8 and Figure 9. Each face that was bolted together got a thin layer of a clear silicone to seal the chamber air tight. Once the silicone dried, the agitation system was installed into the chamber.
The main structure of the RBC unit screwed together is shown in Figure 10. A drill with a Philips bit helped the assembly go faster. The exterior door switch and latch were also installed. From here, custom rubber gaskets were made. These gaskets can be seen in Figure 11 and are used to create an air tight connection from the interior chamber to the outer structure. They were cut out to size with scissors and a hole punch was used to cut out circles for the mounting screws.
Now that the interior chamber and outer box are assembled, they were put together as shown in Figure 12. The rubber gaskets can be seen where the inlet and outlet of the heating system are to be installed. The interior chamber is held inside the outer structure with 8 small bolts. These bolts suspend the interior chamber inside the outer structure with a 1 millimeter air gap between the two.
The next step in the assembly was installing the heating system that was shown in Figure 7. This heat duct bolts inside the rear part of the main structure. As seen in Figure 13, long bolts attached some protective grates, the fan, and the duct system to the RBC unit. More of the rubber gaskets were used to mount the air duct to prevent any leak of hot air. Figure 14 shows a rear view of the heat duct installed and a view of the mounted heat element.
Figure 13: Heating System Installed in Main Structure
In order to seal the top of the structure and allow access to the RBC bags, a door was built. The top panel and door assembly is shown in Figure 15. This door is comprised of two layers of polycarbonate separated by 1mm. This door utilizes the same air gap as seen in Figure 12 to help insulate the machine and keep the heat from escaping the system.

The lid assembly shown in Figure 15 was then installed onto the main structure. This completed the build of the main structure, agitation system, and heating system. Figure 16, Figure 17, and Figure 18 show multiple views of the completed structure. The next step in the build is installing all of the
electrical components. These components will be installed to bring life to the machine and connect all of the systems into a working unit.

Figure 16: Attached Top Plate and Lid
Figure 17: Closed Lid and Basket Installed

Figure 18: Front View of Agitation System Installed in the Main Structure
Installation of the Electrical Components
Now that the machine was almost built, the next step was to install all electrical components. Main components including the power supply, LCD screen, and temperature sensors were the first to be installed. As seen in Figure 19, the power supply was mounted on the lower plate in the electrical housing.

![Figure 19: Mounting of the Power Supply](image)

The next component to be installed was the LCD touch screen. In order to mount the screen, four holes were drilled and tapped for mounting screws. By cutting a recess into the back side of the polycarbonate, the LCD screen was able to be mounted flush with the front of the structure. This can be seen in Figure 20.
In order to mount the four small temperature sensors to the basket, small holes needed to be drilled. The basket was mounted onto a mill and holes were drilled. Figure 21 shows these four mounting points. These points were chosen as the best place under to blood bags to make good contact.
Installation of Temperature Sensors
The temperature sensors were installed so that each bag was constantly in contact with two sensors. The four sensors are powered from a 5V bus, and need only two data lines, SDA and SCL, for I\(^2\)C communication. The installation is shown below in Figure 22, where each sensor has a power and ground wire, as well as the data lines attached.
Figure 23 shows an overview of all four sensors wired together. The sensors were fastened to the two plates on both ends of the basket. They were placed near the middle of the metal bars to ensure better contact with the blood bags. The drilled holes for the sensors to be fastened can also be seen in Figure 23.
Finally, the sensors were tested for operation after the installation process as shown in Figure 24. The entire process to install the sensors took several hours.

![Figure 24: All Four Temperature Sensors Fastened Down](image)

**Installation of the power supply and power switch**
The power supply was fastened down in the back-right corner of the electrical compartment. All electrical parts needing power were fully wired to the power supply as shown in Figure 25. Also seen in this figure, is a switch installed on the back of the box. This was simple to install as it required a power and ground connection to turn on and off the power supply.
Connection of sensors to microcontroller

Both the temperature sensors and the door switch were connected from the RBC compartment to the outer housing. Each wiring assembly was connected to the microcontroller board as shown in Figure 26. The microcontroller is outlined in blue in the figure; the connections to the temperature sensors are on the far left top side of the controller. The connection to the door switch is right next to the temperature sensors on the controller.
Connection of Raspberry Pi 3 to microcontroller
The connection from the Raspberry Pi 3 to the microcontroller was made using the ribbon cable shown in Figure 27 below. This ribbon carried 5V and 3.3V power, as well as UART communication and GPIO connections.
Wiring of Motor and Fan
Both the motor and fan are speed controlled via the switching board. In response to a signal from the microcontroller board, the switching board regulates the duty cycle of the motor and fan. As seen in Figure 28 below, the wires from the motor run underneath the inner compartment where the bags sit. All connections from the inner compartment are routed through one hole in the bottom of the unit for simplicity and ease of access.
Figure 29 shows a view of the door switch connected before putting everything back together. The door switch was tested shortly after installing and gave a successful reading.
Finally, Figure 30 shows an overview shot of the motor wired up and the red cable used to connect to the temperature sensors (refer to *installation of temperature sensors* section).

**Hardware Changes**

During the implementation phase, there were several changes made to the electrical hardware design due to unanticipated hardware restrictions and changes made to the mechanical system. The changes are summarized in Table 1.
Table 1: Summary of Changes Made to Hardware Design

<table>
<thead>
<tr>
<th>Change</th>
<th>Change Made</th>
<th>Reason for change</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2nd Heating Element Added</td>
<td>Time requirement was not met with one heating element</td>
</tr>
<tr>
<td>B</td>
<td>2nd TRIAC Repurposed</td>
<td>To allow switching of current for 2nd heating element</td>
</tr>
<tr>
<td>C</td>
<td>Level Shifter Circuits Added</td>
<td>Difference in voltage levels between Raspberry Pi and ATmega</td>
</tr>
<tr>
<td>D</td>
<td>Hardware Reset Added</td>
<td>To enable hardware reset of control board in case of fault. Required level shifting circuitry.</td>
</tr>
<tr>
<td>E</td>
<td>Motor Switching Transistor Added</td>
<td>To switch motor as TRIACS can only switch AC</td>
</tr>
<tr>
<td>F</td>
<td>Fan Power Transistor Added</td>
<td>To turn fan off. Failed to realize fan chosen still runs at 0 duty cycle</td>
</tr>
<tr>
<td>G</td>
<td>10μF Charge Pump Capacitor</td>
<td>To allow running of motor at lower speeds by smoothing pulse train voltage applied</td>
</tr>
</tbody>
</table>

Change A: 2nd Heating Element

After the initial build of the system, the fan and heater were both run at full capacity to verify the capability of the system to meet time requirements. Heating from 34°C to 41°C took over 20 minutes, which was well outside the specification of 10-15 minutes. The team discussed other options, such as improving isolation between the heating element and the duct, but as another heating element had already been purchased, the decision was made to add this to the design.

Change B: 2nd TRIAC Repurposed

After it was decided to incorporate a second heating element into the design, the current capacity of the VO2223A TRIACS had to be revisited. From their datasheet, the maximum current rating is 1 A RMS. Each heating element is purely resistive and rated for 75 W at 120 V RMS. Therefore, each element sources 0.625 A RMS, so two TRIACS would be needed to safely switch current for two elements. Due to Change E, the TRIAC originally designed to switch the motor was now available. The schematic, and consequent board layout, were modified to repurpose the motor’s TRIAC to drive the second heater.

Change C: Serial Level Shifter Circuit

In the initial design, we did not account for the difference in voltage levels between the serial UART ports on the Raspberry Pi and the ATmega328p. The Raspberry Pi runs on 3.3 V logic level, while the ATmega runs on 5 V. To account for this, two circuits were added.

To transmit from the ATmega to the Raspberry Pi a resistor divider was used, as shown in Figure 31. The values were chosen to limit current and ensure the output to the pi is two-thirds of that transmitted:

\[5V \times 2k(1k+2k) = 3.33V\]
It is important to note that this divider circuit is not the most optimal that could be used. At high switching speeds, parasitic inductive and capacitive effects come into play that introduce data loss. As noted in the testing section, the highest baud rate successfully tested with the system was 19200. This is likely due to this divider circuit. We decided not to improve the circuit because it is sufficient for the purposes of this project. Even with this data rate, the microcontroller is capable of sending the status of all sensors, switches, and buttons every 30ms. As a result, other methods of level translation would only add unnecessary cost and complexity. In the future, if the response time of the system needs to be improved, this is one area of the design which can be improved.

![Figure 31: ATmega to Pi Level Shifter](image)

To transmit from the Pi to the ATmega, two 2n2222a NPN transistors were used, shown in Figure 32. The first transistor converts the transmitted signal to a 5V logic level, but inverts it. The second transistor inverts back to complete the conversion. The resistors chosen limit the current used and ensure transistor saturation in the ON state.

![Figure 32: Pi to ATmega Level Shifter](image)

**Change D: Hardware Reset Added**

During the implementation of the system, some bugs were found that interfered with proper communication between the two controller boards. While those bugs were fixed, it seemed prudent to
add a hardware reset to allow better recovery from any bugs that may not be uncovered in our testing process. To enable the ATmega, its RESET pin is held HIGH with a pull up resistor. A transistor was added to this pull up circuit to allow the RESET pin to swing LOW when the transistor is on. For simplicity, this transistor is the same NPN transistor used in the level shifter circuit. A schematic of the Hardware Reset circuit is shown in Figure 33.

![Figure 33: ATmega Hard Reset Circuit](image)

**Change E: Motor Switching Transistor Added**

In the process of testing motor control with the TRIAC, we realized we had made a fundamental oversight: TRIACS only work with AC loads. Since a zero-voltage crossing will never occur once the TRIAC has been enabled with a DC load, the TRIAC will latch on. While acceptable for some situations, it was necessary to be able to toggle a DC load on and off. In place of a TRIAC, a high voltage, n-channel MOSFET transistor (IRF630) was added to switch the motor. The transistor used was one that we had on hand, and could easily substitute. The final motor switching circuit is shown in Figure 35 under Change G, with the additional charge pump capacitor added.

**Change F: Fan Power Transistor**

While testing the fan, we realized we made an oversight when examining its datasheet. That is, the fan still runs at several thousand rpm without any duty cycle applied. Since it did not seem prudent for reliability of the fan and noise of the system for the fan to be constantly running, an additional IRF630 was added between the fan and ground to allow switching of 12VDC power to the fan. The switching circuit is shown in Figure 34.
Figure 34: Fan Power Control Circuit
Change G: 10uF Charge Pump Capacitor
During testing of the motor, it was determined that driving it at a full 12VDC created an agitation speed faster than what was desired. Using a variable DC supply, it was determined that 9VDC was ideal voltage at which to drive the motor. To resolve this, the duty cycle applied to the motor switching transistor was decreased to an average voltage of 9VDC. However, due to a pulse train now being applied to the motor, its output torque was degraded. At high torque portions of the agitation cycle the motor would approach stall conditions. To solve this, a 10uF capacitor was added across the motor to act as a charge pump and keep the voltage up when the transistor was switched off. The final motor control circuit is shown in Figure 35.

![Figure 35: Final Motor Control Circuit](image)

Design of Microcontroller PCB
To better facilitate the gathering of data from the system’s sensors and devices, a Microcontroller PCB was developed. The purpose of this board is to be the primary link between the Raspberry Pi and all of the systems hardware. A serial link provides the necessary communication between the Microcontroller PCB and the Raspberry Pi. This board primarily includes an ATmega328p MCU and the required clocking hardware, as well as connectors for all low voltage wiring harnesses. Since the ATmega328p operates on 5V logic while the Raspberry Pi utilizes 3.3V logic, level translation hardware was added to this board, corresponding to Change C. Change D, a reset transistor, is included on the board as well as the various pull-up and pull-down resistors required to gather data from buttons and switches. A full schematic of this board is shown in Figure 36.
After the schematic was reviewed and validated, a single-layer layout of the board was developed. While forcing this board to be on a single layer caused the physical footprint to be enlarged, it also allowed the board to be rapidly prototyped using the IPFW ECE department’s ProtoLaser 200 PCB etcher. This laser etcher is capable of creating PCBs from bare copper circuit board material. For this design, the chosen material was ½ oz. Copper Clad 60mil FR4. Presented is the layout and etched version of this PCB in Figure 37 and Figure 38 respectively.
While the ProtoLaser 200 is quite adept at cutting traces, it is not capable of drilling holes for the many through hole components. A micro drill press was used to precisely drill holes in the board. Once etched and drilled, the PCB was tinned to prevent oxidation of the bare copper traces. Tinned copper also bonds with solder easier than bare copper, thus making manual soldering of the PCB much easier. Shown in Figure 39 is an image of the PCB in the tinning solution. Notice the board has become silver in appearance.
Design of Switching PCB
Due to the electrical requirements of the system’s heaters, fan, and motor, another PCB was designed to increase isolation between the communication hardware and the high voltage needed to drive these components. Two optically coupled TRIACS were selected as switching elements to control the 120 VAC heaters. Also known as solid state relays, these components provide a high level of isolation between the logic signal that enables them, and the load. Two MOSFETs are also used to enable the lower power, 12 VDC, fan and motor. A full schematic of the Switching PCB is shown in Figure 40.
The fabrication of this PCB followed the same process used to create the Microcontroller PCB. The board was etched, drilled, and tinned. A full layout of the board is shown in Figure 41. Also shown in Figure 42 are the final soldered versions of both boards.
Software
In the preliminary design of the system, a high-level specification of the software flow was laid out. This specification has been fleshed out and expanded upon throughout the implementation process. The final design of the software is described here. It is divided into four sections:

1. Communication Protocol
2. ATmega Hardware Control Software
3. Pi Controller Software
4. User interface

1. Communication Protocol
As specified before, the design of the system is to offload all I/O control of hardware to the ATmega control board, since the Raspberry Pi GPIO is incapable of sourcing the needed current. Doing this requires a method for sending commands and receiving status updates from the ATmega. As the design of the remaining control software heavily relies on how this works, this is the first step in designing the system architecture.

The first decision to make is whether or not the communication protocol will operate via polling or interrupts. While it is typically better practice to use interrupts to monitor hardware state changes, we chose to poll the ATmega for a few reasons:

- The Raspberry Pi is already much more powerful than an application like this needs. Polling the ATmega a dozen to a hundred times a second will not tie up the processor.
- Using interrupts would require ISRs for every single I/O pin, complicating the program.
- With interrupts, what if multiple events happen simultaneously? Say a button is pressed at the same time the heater is adjusted, while a timer ISR retrieves a new temperature value. The Raspberry Pi's UART FIFO buffer is only 8 symbols deep, there is significant risk of overflowing it if these updates are sent successively.
Most importantly, the controller will need new temperature data in a smaller time interval than that to detect a button press or any other state change. Given that the system knows it needs this information, polling is a more efficient method of collecting all the information it needs (vs sending several small messages).

The commands that the pi needs to send are:

- Status Request
- Fan Power Set
- Fan Duty Set
- Motor Duty Set
- Heater Duty Set
- PWM Frequency Set

A byte is needed to specify the command, and a byte is needed to specify the value of that command. So, a command packet is 2 bytes. Since the system is polling, every command, even those that configure an output, should return a status update for all of the hardware. To keep the size of the data packet as small as possible, the status of each I/O device is stored as a byte. The exceptions are the temperature readings which are floats that must be sent as 4 bytes. There must also be a way of signaling to the Raspberry Pi that the command was received and executed successfully. This is done by sending a NACK, if it is not, or by returning the command type, if it is. The length of a NACK returned packet is 1 byte. If the command is received and executed, the returned packet is a status update 26 bytes long.

The communication protocol should be asynchronous. That is, be able to send messages without having to be synchronized by an external clock. The serial UART ports on both microcontrollers are extremely capable for this form of communication, and so they were chosen as the communication interface. Since the serial UART ports receive/transmit data in bytes, we define a BEGIN byte and an END byte according to that laid out by ASCII. That is:

BEGIN = 0x02
END = 0x03

There is now the issue of these control characters appearing inside the packet. To account for this, the packet is encoded by setting bit 7 high of any byte that is equal to a control character, and inserting an escape character (ESC = 0x1B) before it to signify it has been encoded. If the escape character itself appears in the packet, the same action is performed. The receiving end then performs the appropriate decoding. Since most of the status bytes are Boolean values, there is no concern for them holding the value of a control byte, however all temperature values and duty cycle values (21 bytes) have this potential. Assuming a worst-case scenario where two thirds of these values are equal to control characters, the maximum packet size is now 40 bytes.

The protocol should also have a way of detecting errors. Cyclic Redundancy Check (CRC) is an efficient algorithm that can detect any single-bit and double-bit errors so long as the size of the CRC and the packet size meet Equation 1 below:
Maximum packet size = 2^r-1 where r = degree of the generator polynomial (Eq 2)

Using a checksum of one byte (CRC8) with a polynomial of r=9, the maximum packet size = 64 bytes. With a maximum packet size of 40 bytes, this is more than sufficient for detecting any 2-bit errors.

The transmission packets are shown in Figure 43.

![Figure 43: Serial Communication Packet Structure](image)

2. ATmega Hardware Control Software
The purpose of the ATmega software is to handle all hardware I/O operations. The ATmega was chosen because the Arduino bootloader can be loaded on it allowing it to run Arduino sketches. Arduino sketches are advantageous because they include libraries that abstract operations like reading/writing pins to single line function calls. It also allowed us to test the software/hardware on an inexpensive Arduino development board well before we had the final Microcontroller board etched and assembled. The Arduino language is for the most part identical to C++; however, it does not allow for dynamic memory allocation, and all code runs a startup function followed by a super loop. These considerations were important as maximum container sizes had to be determined by worst case conditions and initialization had to be handled a bit differently than in C++.

The class diagram, in Figure 44, shows the data types used and their functions/attributes. The system has one instance of bWHardware which is a wrapper class for all the hardware in the system. It has an instance of:

- piComm - a class that handles all serial communication with the Raspberry Pi. This includes receiving/decoding commands, sending/encoding ACKs and NACKs, calculating/comparing checksums, framing/deframing transmissions.
- outputControl - a class that operates the fan, heater, and motor, and returns their current status. More specifically, changes/tracks the duty cycle for all three components, the power for the fan, and the pwm frequency for the heater.
- safetySensors - a class that initializes and returns the status of all safety sensors. This includes the door switch and pressure switches. Pressure switches test for a threshold corresponding to the weight of the bags. Unfortunately, testing revealed that the weight of the bags was too distributed for the sensors to measure.
- userInput - a class that initializes and returns the status of all input push buttons. This includes the up, down, select, and back buttons.
- tempSensors - a class that initializes/communicates with the temperature sensors and retrieves the current temperature. Utilizes the Adafruit_MCP9808 temperature sensor class. Sensor readings are stored as a special datatype binaryFloat, which is a union of a byte array and a float that allows the data bytes comprising the float to be placed in a status update packet.

The flow of the program is modeled in the state chart shown in Figure 45. The Arduino start function initializes all hardware through the blood warmer Hardware instance. In the Arduino loop, the blood warmer Hardware instance continuously runs respond(), which checks the serial port for data, and uses an instance of the piComm class to process it. If the data does not start with a BEGIN character, or if the computed checksum does not equal the checksum received, then a NACK is sent after calculating its checksum and assembling it into a packet. Otherwise, the incoming command is loaded into a receive buffer, decoded, and extracted into the piComm object. If the command specifies a change in output, it is passed on to the output hardware object and executed. The status of all hardware is then fetched from its corresponding instance and then sent after encoding it, calculating its checksum, and assembling it into a packet.
3. Pi Controller Software

All front-end and control software for the IRBCW is run on the pi, and was coded using Python 2.7. After determining the communication protocol between the two controller boards, the next big decision influencing software architecture was the library to use for the gui. PYQT4 was chosen as it is a full-featured and well documented library. The general architecture for the program followed the common Model-View-Controller design pattern. To prevent blocking of the user interface while executing the control loop and pinging the hardware control board, the program was split into two threads: a gui thread and a controller thread.

The class diagram in Figure 46, shows the classes comprising the threads and their attributes as well as functionalities. The main (GUI) thread holds the view, which is comprised of the main window, a display timer, and all popups used in the user interface. The view is updated by the controller using PYQT’s signals and slots. These can be thought of as transmitters and receivers, respectively, which allow a thread safe form of intercommunication as all events are posted to a thread queue. The controller itself holds the model of the system hardware, which has a communication link for sending commands to and receiving status updates from the Atmega control board. The serial communication class operates essentially the same as the one on the Atmega.
Figure 46: Class Diagram of Pi Software

Figure 47 shows a high-level flow of the system controller. At system startup, the hardware model/serial link is initialized, and the control timer is set at an interval of 30ms to begin retrieving...
updates from the Atmega. Slots are used to receive save enable, new set temperatures, and system start/stop signals from the UI. The system handler allows starting/stoping the system from any possible state. The update timer callback function does the brunt of the control work. If the system is not running, it fetches updates for the hardware model. When the system is running, it logs the average bag temperature and the current date and time if saving is enabled. It then computes the error by subtracting the current average temperature from the set point temperature. The state of the system is adjusted if needed, and the error is multiplied by a proportional constant to get the required duty cycle to run the heater at. This proportional constant was experimentally determined. The heater is then adjusted and a check is done to see if incubation has completed. On every command sent, a hardware update is received. This update is loaded into the model and all error checks and necessary input actions are executed.

![State Chart of System Controller](image)

**Figure 47: State Chart of System Controller**

4. User Interface

The interface to the user is all done using a class called PYQT, a specific library for python derived from the more commonly known QT. Using QT Designer, all interface design is implemented, and PYQT allows for the XML reading and editing of the window. All push buttons and input to the GUI is then handled with pure python. Signals are created to connect the code communicating with the electrical components to the user interface. The interface to the user is contained within its own thread to eliminate jumping of code.

![User Interface Design](image)

**Figure 48: User Interface Design**
Build Challenges and Changes
During the build of this unit there were several things that did not go as planned, but overall it went together fairly well. Below is a list of each of the challenges and changes that occurred during the assembly and machining of the IRBCW unit.

- DC gear motor shaft
  - The first thing that appeared as a problem is that the shaft of the DC gear motor was actually longer than what was built in the solid model. This caused a problem with the way the drive arms work in the agitation system. The alignment of the links was off, so to resolve this problem we simply cut the motor shaft shorter to match the design.

- Purchasing
  - Upon receiving the purchased parts, all components were checked and it was found that the door latch was not the correct part. During the purchasing, our parts list accidentally had the wrong part number. The correct latch was ordered and replaced.

- Receiving
  - Upon receiving the polycarbonate pieces from the supplier, it was found that there was a missing piece. There was a quantity of two pieces to be made but we only received one. This piece was a simple square that we could quickly machine ourselves. We decided to go ahead and machine this piece so that we could immediately move on to the initial testing of the machine.
- Polycarbonate
  - The final part in the build was dealing with the polycarbonate. Due to the pieces being machined on a CNC router table they came in very true and to size, but what we missed during the design stage is that ordering \( \frac{1}{4} \)" thick polycarbonate means you’re ordering that nominal size. This means that the sheets of \( \frac{1}{4} \)" thick polycarbonate actually vary from 0.190" to 0.250" thick. This made a couple connections in the unit a bit off but not enough to drastically affect the assembly of the unit.

- Installing the LCD screen
  - After mounting the LCD screen, it was found that the HDMI port lined up where two of the push buttons were to go. This meant that the push buttons could not be installed due to the HDMI cable being in the way. To solve this problem these two button locations were shifted down below the cable, as seen in Figure 49. New holes were machined and a cover plate was made to hide the old holes.

![Figure 49: Installed LCD Screen](image)
Section II: Testing and Evaluation
The purpose of this section is to summarize the required testing and results for the Intra-Operative Red Blood Cell Warmer. The several tested listed under this section were designed to fully vet all of the requirements and specifications, given parameters, limitations and constraints, and safety/economic/environmental considerations.

Requirements and Specifications
These include the specific limits and tolerances of the system.

Door Switch
The following test verifies that the door switch is working and ensures that safety measures are working for the operator.

- **Test**: Verify door switch shuts off the system in all possible cases.
- **Procedure**:
  1. Run the system.
  2. Allow system to run for a minute.
  3. Open door abruptly.
  4. Verify system stops.
  5. Prompt user to either continue running or restart.
  6. Does not close prompt till door is closed.
  7. Check that if user clicks continue, the system continues where it left off.
  8. Check that if user clicks restart, the system restarts.
- **Results**: Opened the door and the system shutdown. Then a warning pop-up displayed on the screen as required. When the user clicked “Continue”, the system continued from where it left off. The time also did not restart and continued. The door was opened again to check that the warning pop-up restart button worked. When clicking the restart button, the time did restart. The door was shut one last time and opened. Keeping the door open and clicking continue or restart on the popup did not result in anything happening. This verified that the system would not run until the door closes.

Heating System Verification
The test verifies the system as a whole works including the software. This test is used to modify and tune the control loop within the code.

- **Test**: Verify that the system is heated up to desired temperature within 10 to 15 minutes, and that it incubates for an hour to +/- 1°C.
- **Procedure**:
  1. Insert a wire from the thermocouple inside each bag (tested with colored water).
  2. Close the door with the bags in the system.
  3. Start the system and measure every minute a change that occurs with both the sensors being tested, and the thermocouple readings for each bag.
  4. Record the measurements for one hour when in the incubation period.
5. Use the data to create a graph and verify the sensors are accurate.
6. Repeat these steps several times to ensure consistency

- **Results:** The system was thoroughly tested to determine if it met the requirements and specifications as described in the Problem Statement. Of these, the primary task was to hold a given input temperature for at least an hour. While the system performs up to par under those initial conditions, it fell short when having to raise the temperature up from the lowest possible input temperature to the highest output temperature in the specified time. The following outlines the process used to attempt to resolve this issue and determine its cause. To do this, two 37°C, 400ml, bags of water were placed in the chamber for heating and incubation at a target temperature of 41°C. A calibrated thermocouple was inserted into each bag to monitor the internal water temperature. These values were logged, along with the average temperature as recorded by the four digital temperature sensors. The results of this initial test are shown in Figure 50.

![Figure 50: One Hour Incubation Response 41°C Target](image)

According to the digital temperature sensors, the water reached the desired temperature range within 9.5 minutes. After reaching the desired temperature, the water was incubated within the target temperature range for an additional 60 minutes successfully. It can be noted that the response recorded by the digital temperature sensors matches the characteristic response of a proportional controller. Unfortunately, the temperature recorded by the digital sensors did not accurately match the temperature inside the bags of water until the system reached equilibrium (49 minutes into the test).
Because the sensors were located on the exterior of the RBC bags, it was assumed that the recorded temperature would be affected by the surrounding air temperature, and would subsequently have a faster response time than the blood which has a high specific heat capacity. To calibrate for this difference, an equation was derived mapping the response of the sensors to the temperature of the RBC bags. Since the system ran at full heat capacity for the first 10 minutes, this data was chosen to create the calibration equation. This detailed portion of data can be viewed below in Figure 51.

![Figure 51: Data Used to Create the Calibration Equation](image)

The calibration equation, $C$, was created from the recorded discrete measurements so that the following equation would be correct.

$$T_{Blood} = T_{Recorded} \times C$$

As the constant is dependent on the time spent heating, it was plotted over a 10-minute interval. A cubic trend line of this plot was then used as the calibration equation. This can be viewed below in Figure 52.
The same test was then repeated using the calibration equation; however, the initial conditions for the system were not exactly the same. The input water temperature was set to only 35.8°C, to expose any errors in the equation. It was anticipated that the compensation built into the equation might cause the blood temperature to overshoot if the initial conditions were not the same as they were during calibration. The results of this test can be viewed below in Figure 53, where it is clear that an overshoot indeed occurred.

**Figure 52: The Calibration Equation**

**Figure 53: Response with Calibration**
In attempt to get closer to the target response, the proportional controller constant was maximized. Until this point, the P-Controller operated with a constant that was determined analytically by running the heater at full duty cycle and turning it off at different temperature differences to see which minimized the offshoot. Based off this, the initial constant was chosen to dial the heater down around this point, and then tuned further experimentally. This setup was tested with an input temperature of 35°C. The results of this test can be viewed below in Figure 54.

![Figure 54: Response with Elevated Heating Constant](image-url)

Again, with this test, the recorded temperature from the digital sensors reached the 41°C target within 10 minutes. At this point, the proportional controller would attempt to hold the temperature constant; as a result, heating after this point would be minimal.

Another test was performed in the same fashion; however, the system was preheated for 5 minutes before water was placed in the basket for heating. The results of this test were worse than previous attempts. As anticipated, the digital sensors came up to the target temperature within 6 minutes of the water being added, and the heat output of the system was accordingly reduced. The results of this test can be viewed in Figure 55.
It was clear from the tests performed that the digital temperature sensors were tracking the temperature of the air, and the steel basket far closer than the blood inside of the bags. It would be possible to account for this discrepancy if the initial temperature of the blood, the current temperature of the air, and the current temperature of the steel basket were known. Unfortunately, it is not reasonably possible to measure the temperature inside the bags without contaminating the blood at each system startup with the current available blood bag selection. Ideally, the temperature could be measured by inserting a calibrated thermocouple inside each bag through an isolated measurement port. Currently this sort of bag is not available, making exterior measurements the only way to analyze the temperature of an RBC unit.

Without precisely knowing the initial temperature of the blood, it is not possible to heat the system to a 41°C target temperature without risk of overheating the RBC units. In several tests, our system has proven itself to be capable of tracking a target temperature for an extended period of time, as shown by the blue curve in Figure 50. It is also capable of heating blood from 37°C to 41°C within 15 minutes as shown in Figure 56; however, it cannot do both of these tasks based solely upon the temperature reported by its digital temperature sensors.

**Figure 55: Response with 5 Minute Preheat**
Buttons
The test verifies that all four buttons next to the LCD operate correctly. Two buttons allow the user to go up and down. The tab button will move the focus on different elements within the GUI. The last button should act as a select button.

- **Test:** Verify that all the buttons operate accordingly. Test that the select button operates on all GUI components, and the correct operation occurs when this action is performed. Test that the tab button operates accordingly on all components that should take focus. Test the up and down buttons on the adjusted temperature box when it has focus.

- **Procedure:**
  1. When the program opens up, use the tab button to loop through all components on the GUI that take focus.
  2. Then end focus on the start button again, and use the select button to start the system.
  3. Then use the select button again to stop the system and use the tab button to take focus on the adjustable temperature box.
  4. Use the up button to take the desired temperature to 41°C.
  5. Verify the down button by moving the desired temperature back down to 40°C.
  6. Use the tab button to take focus on the start button again and press the select button to start the system.
  7. While the system is running, open the door and answer the warning popup by using the tab and select buttons.
8. This concludes the procedure for the testing of the buttons and all previous steps should be verified in the results.

- **Results:** Following this procedure, it was seen that the select button works on the start, save, and shutdown buttons. Meaning, when shutdown was selected, the system shutdown, and when the start button was selected, the system started. The tab was also verified to work for all focused properties on the GUI both when the system was running and when it was not running. This test is still to be verified as successful. The up and down buttons both worked as should when the focus was on the adjustable temperature box.

**Agitation**
Blood is sensitive to pressure, which should be taken into consideration when building the system. Too much agitation will damage the red blood cells. The test verifies that the agitation is appropriate.

- **Test:** Verify the agitation system operates at a rate of 10 cycles per minutes (or 6 seconds per cycle).
- **Procedure:**
  1. Plug RBC incubator in
  2. Install two RBC bags
  3. Close lid and select an incubation program and start
  4. Using a stop watch record cycle time multiple times.
- **Results:**
  1. After testing it was found that the system averaged 6.01 seconds per cycle.
  2. Table 2 shows the results.

<table>
<thead>
<tr>
<th>Test</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.90</td>
</tr>
<tr>
<td>2</td>
<td>6.22</td>
</tr>
<tr>
<td>3</td>
<td>5.81</td>
</tr>
<tr>
<td>4</td>
<td>6.22</td>
</tr>
<tr>
<td>5</td>
<td>5.91</td>
</tr>
</tbody>
</table>

**External Case Temperature**
External case temperature should not be higher than 50°C, so as to not inflict physical harm. The test verifies the external temperature of the device.

- **Test:** Verify the exterior is always less than 50°C during an entire incubation cycle at the maximum allowed temperature.
- **Procedure:**
  1. Pre-warm bags to 37°C
2. Plug RBC incubator in and select a desired temperature of 41°C
3. Install the two blood bags
4. Close lid and start program
5. Run the incubation cycle for one hour
6. Using a thermocouple tape the end to different points across the exterior
7. Make sure to test key points
   a. Above heating duct
   b. Door handle
   c. Front panel
   d. Push buttons
   e. LCD screen
   f. Above heat duct outlet
8. Record results

- **Results:**

<table>
<thead>
<tr>
<th>Location</th>
<th>Temperature (°C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Above Heating Duct</td>
<td>30.7</td>
</tr>
<tr>
<td>Door Handle</td>
<td>27.8</td>
</tr>
<tr>
<td>Front Panel</td>
<td>29.5</td>
</tr>
<tr>
<td>Push Buttons</td>
<td>30.3</td>
</tr>
<tr>
<td>LCD Screen</td>
<td>28.3</td>
</tr>
<tr>
<td>Above Heat Duct Outlet</td>
<td>30.1</td>
</tr>
</tbody>
</table>

**Weight and Stability**
The system should be compatible with standard IV poles and the total weight shall not exceed 16Kg. The test verifies the weight of the unit.

- **Test:** Verify that the machine clamps onto a standard IV pole and is stable.
- **Procedure:**
  1. Using a scale, weigh the RBC unit
  2. Mount the unit to the IV pole and tilt the base slightly to ensure stability
- **Results:**
  1. Weight of the unit is 10.3 kg

**Heat Loss**
Minimize system heat loss to maximize efficiency.

- **Test:** Monitor and calculate heat loss to surroundings
- **Procedure:**
  1. Plug RBC incubator in and select a desired temperature of 41°C
  2. Install thermocouple to the interior wall of the heat chamber
  3. Close lid and start program
4. Run the incubation cycle for one hour
5. Monitor and record the interior and exterior wall temperature
6. Calculate heat loss from system using the following information and equations

\[ T_{\infty,i} = \text{internal wall temperature} = 42 \, ^\circ\text{C} \]
\[ T_{\infty,o} = \text{external wall temperature} = 20 \, ^\circ\text{C} \]
\[ L_1 = L_3 = \text{wall thickness} = 0.00635 \, \text{m} \]
\[ L_2 = \text{air gap thickness} = 0.0015 \, \text{m} \]
\[ k_1 = k_3 = \text{thermal conductivity of walls} = 0.19 \, \text{W/}m \cdot \text{K} \]
\[ k_2 = \text{thermal conductivity of air gap} = 27.0215 \times 10^{-3} \, \text{W/}m \cdot \text{K} \]
\[ A_T = A_B = \text{area of top and bottom of chamber} = 0.061935 \, \text{m}^2 \]
\[ A_S = \text{area of sides of chamber} = 0.180645 \, \text{m}^2 \]

\[ q = \frac{T_{\infty,i} - T_{\infty,o}}{L_1 + \frac{L_2}{k_2} + \frac{L_3}{k_3}} = 7.5192 \, \text{W/m}^2 \]

\[ Q = (A_T + A_B + A_S)q = 2.2897 \, \text{W} \]

- **Results:** The heat loss from the system was found to be 2.3W which is better than the 8.1W that were calculated last semester.

**Communication Protocol Test**

To determine the maximum polling speed of the system, a short program was run to test the communication software. The program was used to test the system at baud rates of 9600, 19200, 38400, 57600 by sending 1000 transmissions and counting the number of responses successfully received and the average time they took. The system was then optimized based on this info. Results of the test are seen below in Figure 57. After tuning, the average transmission ping time was 25.6ms, with zero failed responses. Due to this result, the system polling time was set to 30ms, which is sufficient for the system. The maximum baud rate achieved was 19200, no responses were received at higher rates. This is likely due to the resistor divider circuit detailed earlier, and could be improved by replacing it with a switched level shifter.

<table>
<thead>
<tr>
<th>Baud Rate:</th>
<th>19200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages Sent:</td>
<td>1000</td>
</tr>
<tr>
<td>Responses Failed:</td>
<td>0</td>
</tr>
<tr>
<td>Responses Received:</td>
<td>1000</td>
</tr>
<tr>
<td>Total Time Elapsed:</td>
<td>157.35807395</td>
</tr>
<tr>
<td>Average Ping Time:</td>
<td>0.157486291885</td>
</tr>
</tbody>
</table>
Hardware Control Testing

During development, control of the hardware was tested from the Pi side of the communication link throughout development using another test program, seen below in Figure 58. The program updated the status of all inputs and outputs at the specified polling rate, and allowed output commands to be sent to the Atmega. Using this, the AtMega I/O was first tested by probing appropriate pins on an arduino loaded with the control software using a volt meter and comparing it to the results.

After everything tested to be working correctly, the I/O hardware was then connected to the Arduino using a breadboard and tested to make sure it could be controlled properly, see Figure 59 below. At this point, some hardware and software changes were made. Once everything worked properly, the controller software could then be tested. When the final control/switching boards were fabricated and assembled, the program was again used to verify/troubleshoot proper operation of components.
Testing Challenges and Changes

When the machine was up and running, several things came to light that needed to be changed. Multiple failures and system bugs prevented the machine from working during the first start-up. The following is a list of all the challenges and changes that occurred during the testing of the RBC unit. After addressing these issues, the RBC unit started working great and allowed the final testing to occur.

- Adding a heat element
  - Once all the major components were installed, it was time to do some initial testing. The first test was to determine if the heat element selected was powerful enough to bring the blood bags up to temperature within the desired time. According to the requirements, the bags needed to go from 37°C to 41°C in 10 to 15 minutes. To test this, the heat element was hooked up to 120V AC and the fan was powered to 12V. A thermocouple was mounted to the bags and temperatures were recorded. It was found that it took 23 minutes to go from 35°C to 42°C. These temperatures were chosen as an extreme case to insure the system could handle any situation close to what was desired. These results showed that the required heating time was not achieved. To combat this, it was decided to add an extra heat element. Once mounted in the system, the test was repeated and it was found that the new setup took 13 minutes to go from 35°C to 42°C.

- Air diffuser plate (Aluminum)
  - Once the heat elements were dialed in, the next change came to the hot air flow. During the initial heating of the bags, it was found that the air coming out of the heat duct was roughly 60°C. This is enough heat to damage the blood. In order to ensure the leading edge of the bags would not get too hot, it was decided to install an air diffuser plate. The diffuser plate was designed as shown in Figure. This plate blocks the flow of air from directly hitting the front edge of the bag. Once installed and tested, it was
found that it took 17 minutes to go from 35°C to 42°C. This did not pass the time requirements and was due to the increased amount of material that the heat had to sink into before the bags could start their temperature climb. From here, it was decided to move the diffuser to the basket in the form of a small polycarbonate wall.

- Air diffuser plate (Polycarbonate)
  - Initially, the air diffuser plate was designed as a rectangle polycarbonate piece. This piece was mounted to the agitation basket and the heat up test was preformed again. The results got the rise time in the 10-15minute window desired. After monitoring both bags, it was found that there was a 0.5°C difference in temperature between them. This was a result from one bag getting more air flow than the other. The pivot point of the agitation system is not aligned with the hot air duct. To solve this, an angled diffuser plate was designed as seen in Figure.
• Broken wires
  o With the electrical system installed and up and running, the first data test was started. In this test the system was ran for an hour. The agitation system had four temperature sensors attached to the basket. These temperature sensors have a connection under the basket that failed. The failure was because the connection was not secured properly. To fix this problem the connection and wires were secured to the basket. This kept the connection from moving and forced the actual wire to bend.

• Catastrophic melt down
  o As stated in the broken wire section, the temperature sensor wires broke. This caused the heat elements to overrun. As seen in Figure, the point where the heat duct bolts to the housing melted. This required a redesign on the heat duct and programing. The programing was re-evaluated to have a safety check. This safety check will monitor the temperature sensors and will shut down the system if no signal is found. This programing change will prevent this from happening again.
  o New polycarbonate was machined as seen in Figure
Figure 62: Damage from Overheating
Figure 63: Replacement Wall Piece

- Insulating the heat duct
  - After the melting of the structure, it was decided to address the heat duct. As seen in Figure, new insulation was installed. A high temperature insulation was installed in between the polycarbonate housing and hot heat duct. This will prevent any damage in the future if more problems are to occur.
Figure 64: New Insulation Installed on Heat Duct

- Heat element mounting
  - In addition to insulating the heat duct, some changes were made to the heat elements. The old design of the heat element mounts is shown in Figure 64. This design clamped the heat elements directly to the aluminum mount. This caused an issue of too much energy getting transferred into the heat duct. As a result, the temperature of the bags was hard to regulate. In order to help prevent this situation, the heat elements were mounted differently. Instead of directly clamping the elements to the aluminum mount, the elements were suspended inside the duct. A cross-shaped wire was installed into the duct to suspend the elements. This greatly reduced the heat transferred directly to the duct. Some benefits of this change are a faster heating time and also more precision during the incubation process.
• Adding feet
  
  o With the machine designed to be mounted on an IV pole it was not taken into account that it may need to sit on a table top. During testing, it was decided to add simple rubber feet to the bottom of the machine. The bottom surface of the machine has several screw heads that can scratch the table surface. By adding the rubber feet shown in Figure, the machine can nicely and securely sit on any table surface if desired.
Figure 66: Addition of Rubber Feet
Section III: Cost/Budget
The budget for the Intra-operative Red Blood Cell Warmer project was set by Zimmer Biomet at $5,000. During the first semester of the project it was estimated that it would cost $1,590.66 to purchase all of the parts and components necessary to complete the construction of the prototype. However, the total cost of the prototype came to $4,359.69 which is broken down in Table 4. The large increase in the cost was due to the costs of having the polycarbonate parts, air duct, and agitation components machined. The final cost of the project was still under $5,000 budget.

Table 4: Breakdown of Budget Expenses

<table>
<thead>
<tr>
<th>Description</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchased Parts</td>
<td>$950</td>
</tr>
<tr>
<td>Raw Aluminum</td>
<td>$33.19</td>
</tr>
<tr>
<td>Polycarbonate</td>
<td>$2414</td>
</tr>
<tr>
<td>Sheet Metal</td>
<td>$880.50</td>
</tr>
<tr>
<td>Miscellaneous Parts</td>
<td>$82</td>
</tr>
</tbody>
</table>
Conclusion
After running into major setbacks and complications during the building process and the beginning of testing, the team can finally say that the Intra-operative Red Blood Cell Warmer was successfully designed and assembled. By doing all the tests mentioned above, the incubation unit proved to meet all the requirements and specifications set at the beginning of the design stage except one.

The requirements/parameters that were verified to be achieved are:

- The unit stops incubating in the event the door is opened mid cycle.
- The temperature sensors have an accurate steady state reading of +/- 0.1°C
- Once at the desired temperature, the machine can incubate for an entire hour with +/- 0.3°C accuracy
- The machine does not get too hot on the outside (50°C or more) which could cause bodily injury
- The machine is as accurate as possible by reducing the heat loss of the system
- The machine can operate while being clamped to a standard IV pole during the entire incubation cycle.
- The machine cost is below the budget set by Zimmer Biomet.

As mention in Section II, the only requirement that was not met was the machine could not heat up the temperature from 37 to 41°C in less than 15 minutes. Even after thorough device characterization, this task is impossible without knowing the initial RBC temperature at the bag’s core. Because the unit only reads the surface temperature of the bags, the systems starts the incubation cycle once the outside temperature hits 41°C, even though the inside temperature is not necessarily 41°C. Heat transfer equations could be derived to analyze the amount of heat added to the RBC, but without knowing the initial temperature of the RBC it would be possible for overheating to occur.

To combat this problem, the Senior Design Team recommends Zimmer Biomet to consider the development of a sanitary measurement method to accurately gauge the core temperature of an RBC unit. Currently, no system exists that allows for a core measurement to be taken without contaminating the RBC sample. If this were possible, the core temperature of an RBC unit could be fed to the IRBCW to rapidly, and accurately, heat and incubate two RBC units.

It is strongly believed that the designed process worked really well for the machine since the unit met almost every parameter/requirement with the design that was chosen. Even though the system is unable to reach the target temperature in the desired time, it is capable of accurately incubating blood for over an hour, thus the system meets its primary goal. Figures 64, 65, and 66 show the final version of the IRBCW.

After finally having the unit successfully up and running, it was acknowledged that the design process that was done last semester was very helpful because it made it easier for the team to decide what route to take with every single component (sub-system) in an organized and efficient fashion.
Figure 67: Red Blood Cell Warmer Clamped to IV Pole

Figure 68 and 69: Red Blood Cell Warmer Clamped to IV Pole
References

Appendix A: Test Data

Table 5: Data for Figure 50

<table>
<thead>
<tr>
<th>Time</th>
<th>Recorded Temperature</th>
<th>Bag 1</th>
<th>Bag 2</th>
<th>Desired</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>33.1</td>
<td>36.4</td>
<td>36.5</td>
<td>41</td>
</tr>
<tr>
<td>1</td>
<td>33.5</td>
<td>36.5</td>
<td>36.7</td>
<td>41</td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>36.6</td>
<td>36.9</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>34.8</td>
<td>36.6</td>
<td>37</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>35.6</td>
<td>36.9</td>
<td>37.3</td>
<td>41</td>
</tr>
<tr>
<td>5</td>
<td>36.6</td>
<td>37.2</td>
<td>37.6</td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td>37.6</td>
<td>37.5</td>
<td>37.9</td>
<td>41</td>
</tr>
<tr>
<td>7</td>
<td>38.5</td>
<td>38</td>
<td>38.3</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>39.3</td>
<td>38.2</td>
<td>38.6</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>40.2</td>
<td>38.7</td>
<td>39.1</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>41.1</td>
<td>39.3</td>
<td>39.4</td>
<td>41</td>
</tr>
<tr>
<td>11</td>
<td>41.2</td>
<td>39.4</td>
<td>39.5</td>
<td>41</td>
</tr>
<tr>
<td>12</td>
<td>41.1</td>
<td>39.3</td>
<td>39.5</td>
<td>41</td>
</tr>
<tr>
<td>13</td>
<td>40.9</td>
<td>39.3</td>
<td>39.5</td>
<td>41</td>
</tr>
<tr>
<td>14</td>
<td>40.8</td>
<td>39.3</td>
<td>39.5</td>
<td>41</td>
</tr>
<tr>
<td>15</td>
<td>40.7</td>
<td>39.4</td>
<td>39.5</td>
<td>41</td>
</tr>
<tr>
<td>16</td>
<td>40.7</td>
<td>39.5</td>
<td>39.7</td>
<td>41</td>
</tr>
<tr>
<td>17</td>
<td>40.7</td>
<td>39.6</td>
<td>39.6</td>
<td>41</td>
</tr>
<tr>
<td>18</td>
<td>40.8</td>
<td>39.8</td>
<td>39.8</td>
<td>41</td>
</tr>
<tr>
<td>19</td>
<td>40.8</td>
<td>39.9</td>
<td>39.9</td>
<td>41</td>
</tr>
<tr>
<td>20</td>
<td>40.9</td>
<td>40</td>
<td>39.9</td>
<td>41</td>
</tr>
<tr>
<td>21</td>
<td>40.9</td>
<td>39.9</td>
<td>39.9</td>
<td>41</td>
</tr>
<tr>
<td>22</td>
<td>40.9</td>
<td>40</td>
<td>39.9</td>
<td>41</td>
</tr>
<tr>
<td>23</td>
<td>40.9</td>
<td>40.1</td>
<td>40.1</td>
<td>41</td>
</tr>
<tr>
<td>24</td>
<td>40.9</td>
<td>40.2</td>
<td>40.1</td>
<td>41</td>
</tr>
<tr>
<td>25</td>
<td>40.9</td>
<td>40.2</td>
<td>40.2</td>
<td>41</td>
</tr>
<tr>
<td>26</td>
<td>40.9</td>
<td>40.2</td>
<td>40.4</td>
<td>41</td>
</tr>
<tr>
<td>27</td>
<td>40.9</td>
<td>40.4</td>
<td>40.3</td>
<td>41</td>
</tr>
<tr>
<td>28</td>
<td>40.9</td>
<td>40.4</td>
<td>40.3</td>
<td>41</td>
</tr>
<tr>
<td>29</td>
<td>40.9</td>
<td>40.5</td>
<td>40.3</td>
<td>41</td>
</tr>
<tr>
<td>30</td>
<td>40.9</td>
<td>40.5</td>
<td>40.3</td>
<td>41</td>
</tr>
<tr>
<td>31</td>
<td>40.9</td>
<td>40.6</td>
<td>40.5</td>
<td>41</td>
</tr>
<tr>
<td>32</td>
<td>40.9</td>
<td>40.6</td>
<td>40.5</td>
<td>41</td>
</tr>
<tr>
<td>33</td>
<td>40.9</td>
<td>40.7</td>
<td>40.5</td>
<td>41</td>
</tr>
<tr>
<td>34</td>
<td>40.9</td>
<td>40.7</td>
<td>40.6</td>
<td>41</td>
</tr>
<tr>
<td>35</td>
<td>41</td>
<td>40.7</td>
<td>40.6</td>
<td>41</td>
</tr>
<tr>
<td>37</td>
<td>41</td>
<td>40.7</td>
<td>40.6</td>
<td>41</td>
</tr>
<tr>
<td>38</td>
<td>41</td>
<td>40.7</td>
<td>40.6</td>
<td>41</td>
</tr>
<tr>
<td>39</td>
<td>40.9</td>
<td>40.8</td>
<td>40.6</td>
<td>41</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>40.8</td>
<td>40.7</td>
<td>41</td>
</tr>
<tr>
<td>41</td>
<td>41</td>
<td>40.8</td>
<td>40.7</td>
<td>41</td>
</tr>
<tr>
<td>42</td>
<td>41</td>
<td>40.9</td>
<td>40.8</td>
<td>41</td>
</tr>
<tr>
<td>43</td>
<td>41</td>
<td>40.9</td>
<td>40.7</td>
<td>41</td>
</tr>
<tr>
<td>44</td>
<td>41</td>
<td>41</td>
<td>40.8</td>
<td>41</td>
</tr>
<tr>
<td>45</td>
<td>41</td>
<td>41</td>
<td>40.8</td>
<td>41</td>
</tr>
<tr>
<td>46</td>
<td>41</td>
<td>41</td>
<td>40.8</td>
<td>41</td>
</tr>
<tr>
<td>47</td>
<td>41</td>
<td>41</td>
<td>40.8</td>
<td>41</td>
</tr>
<tr>
<td>48</td>
<td>41</td>
<td>41</td>
<td>40.8</td>
<td>41</td>
</tr>
<tr>
<td>49</td>
<td>41</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>50</td>
<td>41</td>
<td>41.1</td>
<td>40.9</td>
<td>41</td>
</tr>
<tr>
<td>51</td>
<td>41</td>
<td>41.1</td>
<td>40.9</td>
<td>41</td>
</tr>
<tr>
<td>52</td>
<td>41</td>
<td>41</td>
<td>40.9</td>
<td>41</td>
</tr>
<tr>
<td>53</td>
<td>41</td>
<td>41.2</td>
<td>40.9</td>
<td>41</td>
</tr>
<tr>
<td>54</td>
<td>41</td>
<td>41.1</td>
<td>40.9</td>
<td>41</td>
</tr>
<tr>
<td>55</td>
<td>41</td>
<td>41.1</td>
<td>40.9</td>
<td>41</td>
</tr>
<tr>
<td>56</td>
<td>41</td>
<td>41.1</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>57</td>
<td>41</td>
<td>41.2</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>58</td>
<td>41</td>
<td>41.1</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>59</td>
<td>41</td>
<td>41.1</td>
<td>40.9</td>
<td>41</td>
</tr>
<tr>
<td>60</td>
<td>41</td>
<td>41.1</td>
<td>40.9</td>
<td>41</td>
</tr>
<tr>
<td>61</td>
<td>41</td>
<td>41.2</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>62</td>
<td>41</td>
<td>41.2</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>63</td>
<td>41</td>
<td>41.1</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>64</td>
<td>41</td>
<td>41.1</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>65</td>
<td>41</td>
<td>41.2</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>66</td>
<td>40.9</td>
<td>41.2</td>
<td>41.1</td>
<td>41</td>
</tr>
<tr>
<td>67</td>
<td>40.9</td>
<td>41.2</td>
<td>41.1</td>
<td>41</td>
</tr>
<tr>
<td>68</td>
<td>40.9</td>
<td>41.2</td>
<td>41.1</td>
<td>41</td>
</tr>
<tr>
<td>69</td>
<td>41</td>
<td>41.2</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>70</td>
<td>41</td>
<td>41.1</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>71</td>
<td>41</td>
<td>41.2</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>Time</td>
<td>Displayed</td>
<td>Avg Bag 1 &amp; 2</td>
<td>Bag 2</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>---------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>36.3</td>
<td>35.7</td>
<td>35.8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>36.3</td>
<td>35.9</td>
<td>35.6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>36.5</td>
<td>36.3</td>
<td>35.6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>36.8</td>
<td>36.6</td>
<td>35.6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>37.3</td>
<td>36.8</td>
<td>35.8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>37.8</td>
<td>37.2</td>
<td>36.1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>38.1</td>
<td>37.6</td>
<td>36.2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>38.5</td>
<td>38</td>
<td>36.5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>38.7</td>
<td>38.2</td>
<td>36.6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>39</td>
<td>38.8</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>39.1</td>
<td>39.1</td>
<td>37.4</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>39.2</td>
<td>39.5</td>
<td>37.6</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>39.3</td>
<td>40</td>
<td>38.1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>39.4</td>
<td>40.6</td>
<td>38.5</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>39.3</td>
<td>41.1</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>39.3</td>
<td>41.4</td>
<td>39.4</td>
<td></td>
</tr>
</tbody>
</table>
Table 7: Data for Figure 54

<table>
<thead>
<tr>
<th>Time</th>
<th>Display</th>
<th>Bag 1</th>
<th>Bag 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>29.5</td>
<td>35.7</td>
<td>35.2</td>
</tr>
<tr>
<td>1</td>
<td>30.2</td>
<td>35.1</td>
<td>35.1</td>
</tr>
<tr>
<td>2</td>
<td>31.6</td>
<td>35.1</td>
<td>35.3</td>
</tr>
<tr>
<td>3</td>
<td>32.6</td>
<td>35.2</td>
<td>35.4</td>
</tr>
<tr>
<td>4</td>
<td>34.0</td>
<td>35.4</td>
<td>35.5</td>
</tr>
<tr>
<td>5</td>
<td>35.2</td>
<td>35.4</td>
<td>35.8</td>
</tr>
<tr>
<td>6</td>
<td>36.5</td>
<td>35.7</td>
<td>35.7</td>
</tr>
<tr>
<td>7</td>
<td>37.5</td>
<td>36.0</td>
<td>36.2</td>
</tr>
<tr>
<td>8</td>
<td>38.6</td>
<td>36.1</td>
<td>36.4</td>
</tr>
<tr>
<td>9</td>
<td>39.5</td>
<td>36.4</td>
<td>36.9</td>
</tr>
<tr>
<td>10</td>
<td>40.5</td>
<td>36.9</td>
<td>37.3</td>
</tr>
<tr>
<td>11</td>
<td>41.1</td>
<td>37.2</td>
<td>37.6</td>
</tr>
<tr>
<td>12</td>
<td>41.6</td>
<td>37.6</td>
<td>38.0</td>
</tr>
</tbody>
</table>
Table 8: Data for Figure 55

<table>
<thead>
<tr>
<th>Time</th>
<th>Display</th>
<th>Bag 1</th>
<th>Bag 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>27.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>29.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>32.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>37.2</td>
<td>36.9</td>
<td>36.8</td>
</tr>
<tr>
<td>6</td>
<td>36.9</td>
<td>36.8</td>
<td>37.2</td>
</tr>
<tr>
<td>7</td>
<td>37.5</td>
<td>37</td>
<td>37.2</td>
</tr>
<tr>
<td>8</td>
<td>38.3</td>
<td>37.2</td>
<td>37.4</td>
</tr>
<tr>
<td>9</td>
<td>39.3</td>
<td>37.3</td>
<td>37.6</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>37.6</td>
<td>37.7</td>
</tr>
<tr>
<td>11</td>
<td>40.6</td>
<td>37.8</td>
<td>37.9</td>
</tr>
<tr>
<td>12</td>
<td>41</td>
<td>38.1</td>
<td>38.1</td>
</tr>
<tr>
<td>13</td>
<td>41.1</td>
<td>38.2</td>
<td>38.4</td>
</tr>
<tr>
<td>14</td>
<td>41</td>
<td>38.3</td>
<td>38.4</td>
</tr>
<tr>
<td>15</td>
<td>40.8</td>
<td>38.5</td>
<td>38.6</td>
</tr>
<tr>
<td>16</td>
<td>40.7</td>
<td>38.6</td>
<td>38.7</td>
</tr>
<tr>
<td>17</td>
<td>40.7</td>
<td>38.7</td>
<td>38.8</td>
</tr>
<tr>
<td>18</td>
<td>40.7</td>
<td>38.9</td>
<td>39</td>
</tr>
</tbody>
</table>
Appendix B: Raspberry Pi Code

#-*- coding: cp1252 -*-
#----------------------------------------------------------------------------------#
#
# Program Description: Used to control a blood warmer system
# Date Last Modified: December 05, 2016
# File name: main.py -- includes all dependencies
#
#----------------------------------------------------------------------------------#

#-------------------------------------------#
# INCLUDES
#-------------------------------------------#
import sys, time, os
import RPi.GPIO as GPIO
from PyQt4 import QtCore, QtGui, uic
from arduinoComm import arduinoComm
from hardwareState import hardwareState
from controller import controller
from mainWindow import mainWindow
from warningPopup import warningPopup
from errorPopup import errorPopup
from messagePopup import messagePopup

"""
Description: Shutdown event handler, deletes objects, quits app, and shuts os down
Inputs: None
Outputs: None
"""
@QtCore.pyqtSlot()
def shutdown():
    #Close save file
    try:
        controller.saveFile.close()
    except:
        pass
    controller.stopSystem()
    #Clean up objects
    GPIO.cleanup()
    controller.deleteLater()
    window.deleteLater()
    #Exit threads
    controllerThread.exit()
    #Quit app and shutdown raspbian
    app.quit()
    os.system('shutdown now -h')

"""
Description: Connects signals to slots for event handling and relaying information across threads
Inputs: None
Outputs: None
"""
def signalHandler():

84
# Allows controller to send updated, averaged temp values to gui
controller.tempUpdate.connect(window.setTemps)

# Connect start button to its event handler
window.startButton.clicked.connect(window.startClicked)

# Connect set temp adjustment to its event handler
window.adjustTemp.valueChanged.connect(window.setTempUpdate)

# Sends new set temp to controller when changed
window.sendTemp.connect(controller.updateSetTemp)

# Notifies the controller whether to run or not
window.systemState.connect(controller.systemHandler)

# Triggers tactile input event handlers
controller.upPressed.connect(window.upButtonHandler)
controller.downPressed.connect(window.downButtonHandler)
controller.backPressed.connect(window.backButtonHandler)
controller.selectPressed.connect(window.selectButtonHandler)

# popup event handlers
controller.doorSafetyWarning.connect(warning.setWarning)
controller.arduino.systemError.connect(error.setError)
controller.incubationFinishedMessage.connect(message.displayMessage)

# Warning Connections
warning.continueButton.clicked.connect(warning.chooseResume)
warning.restartButton.clicked.connect(warning.chooseRestart)
warning.continueButton.pressed.connect(warning.chooseResume)
warning.restartButton.pressed.connect(warning.chooseRestart)

# system display update
controller.systemUpdate.connect(window.updateStatus)

# display timer connections
controller.startGuiTimer.connect(window.setGuiTimer)
controller.stopGuiTimer.connect(window.stopGuiTimer)

# save button
window.saveButton.clicked.connect(controller.enableSaving)

# Connect shutdown button to its event handler
window.shutdownButton.clicked.connect(shutdown)
window.shutdownPressed.connect(shutdown)

if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    controller = controller()
    window = mainWindow()
    warning = warningPopup()
    error = errorPopup()
    message = messagePopup()
# CREATE THREADS FOR CONTROLLER & SERIAL LINK
#-----------------------------------------------------#
controllerThread = QtCore.QThread()

#-----------------------------------------------------#
# SERIAL LINK AND CONTROLLER SET TO THREADS
#-----------------------------------------------------#
controller.moveToThread(controllerThread)

#-----------------------------------------------------#
# SIGNAL CONNECTIONS
#-----------------------------------------------------#
signalHandler()

#-----------------------------------------------------#
# START HARDWARE AND CONTROLLER THREADS
#-----------------------------------------------------#
controllerThread.start()

#-----------------------------------------------------#
# GUI OUTPUT
#-----------------------------------------------------#
window.show()
sys.exit(app.exec_())

# -*- coding: cp1252 -*-
#-----------------------------------------------------#
# File Description: Used to display Gui to user
# Date Last Modified: December 05, 2016
# File name: mainWindow.py -- includes all dependencies
#-----------------------------------------------------#

#-----------------------------------------------------#
# INCLUDES
#-----------------------------------------------------#
import sys, time, os, pyautogui
from PyQt4 import QtCore, QtGui, uic

#-----------------------------------------------------#
# LOAD GUI FILE
#-----------------------------------------------------#
qtMainWindowFile = "/home/pi/Documents/BloodWarmer/interface.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtMainWindowFile)

#-----------------------------------------------------#
# Class Description: This class is used for the main window.
# Last Edited: 11/17/2016
# Last Edited By: Jon Wesner
# Last Changes Made: 1.0.1:Pete - implemented slot to receive and display temp update
# implemented slots for startButton.clicked and adjustTemp.valueChanged,
functions signal controller for appropriate action, moved to own file
added signaling and templates for button event handlers
1.0.2: Jon - implemented a start, stop, reset gui timer functions and handled within
the startClicked slot. Removed cluttered code unneeded in order to
help read code better. Implemented the button slots using pyautogui
library, which fires off appropriate key strokes based on buttons
pressed.
# ---------------------------------------------------------------------------------------------------------------------------#
class mainWindow(QtGui.QMainWindow, Ui_MainWindow):
    sendTemp = QtCore.pyqtSignal(float)
    systemState = QtCore.pyqtSignal(bool, bool)
    shutdownPressed = QtCore.pyqtSignal()
    guiTimer = QtCore.QTimer()

def __init__(self):
    os.system("xinput set-prop 'Microchip Technology Inc. AR1100 HID-MOUSE' 'Evdev Axis Inversion' 1 1")
    QtGui.QMainWindow.__init__(self)
    Ui_MainWindow.__init__(self)
    self.setupUi(self)
    self.guiTimer.timeout.connect(self.runGuiTimer)

# Local Variables
# ---------------------------------------------------#
self._time = 0
self.systemIsRunning = False
self.targetedTemperature = 0
self.focusedProperty = self.startButton.setFocus()
self.propertyTable = [self.startButton, self.saveButton, self.shutdownButton, self.adjustTemp]

# User Interface Property changes
# ---------------------------------------------------#
self.showFullScreen()  # Fills whole screen depending on screen size
self.startButton.setCursor(QtGui.QCursor(QtCore.Qt.PointingHandCursor))  # Change mouse cursor when hovering over button
self.adjustTemp.setRange(37, 41)
self.adjustTemp.setDecimals(1)
self.adjustTemp.setSingleStep(.1)
self.adjustTemp.setSuffix("°C")

@QtCore.pyqtSlot(bool)
def setGuiTimer(self, restart):
    if restart:
        self._time = 0
        self.guiTimer.start(1000)

@QtCore.pyqtSlot()
def stopGuiTimer(self):
    self.guiTimer.stop()
def runGuiTimer(self):
    self._time += 1
    hours = self._time / 3600
    minutes = (self._time % 3600) / 60
    seconds = self._time % 60
    # Output time to Label
    self.bagTimer.setText('{:02.0f}'.format(hours) + ':' + '{:02.0f}'.format(minutes) + ':' + '{:02.0f}'.format(seconds))

# Description: Event handler for start/stop button. Relays set temp and signals controller to start/stop system
# Inputs: None
# Outputs: systemState - whether control should run or not, setTemp - controller reference temperature

def startClicked(self):
    if not self.systemIsRunning:
        setTemp = self.adjustTemp.text()
        setTemp = setTemp[0:3]
        setTemp = float(setTemp)
        self.sendTemp.emit(setTemp)
        self.startButton.setText("Stop")
        self.systemIsRunning = True
    else:
        self.startButton.setText("Start")
        self.systemIsRunning = False
        self.systemState.emit(self.systemIsRunning, True)

# Description: Event handler for set temp adjustment, sends set temp to controller
# Inputs: None
# Outputs: emits new set temp

def setTempUpdate(self):
    newSetTemp = self.adjustTemp.value()
    self.sendTemp.emit(newSetTemp)

# Function Description: This function is used to send updated data to gui
# Last Edited: 10/13/2016
# Last Edited By: Jonathan Wesner
# Last Changes Made: ...
def setTemps(self, bagTempAvg):
    self.bagTemp.setText("%.1f" % bagTempAvg)

def getButtonFocus(self):
    # length of property table
    lengthOfTable = len(self.propertyTable)
    count = 0
    for propertyInTable in self.propertyTable:
        if propertyInTable.hasFocus() == True:
            self.focusedProperty = propertyInTable
            break
        count = count + 1

def displayError(self, errorText):
    self.errorPopup.errorMessage.setText(errorText)
    # Freezes the Main Window till a response is made by the user for MyPopup()
    self.errorPopup.setWindowModality(QtCore.Qt.ApplicationModal)
    self.errorPopup.show()
@QtCore.pyqtSlot()
def downButtonHandler(self):
    pyautogui.press('down')

@QtCore.pyqtSlot()
def backButtonHandler(self):
    pyautogui.press('tab')

@QtCore.pyqtSlot()
def selectButtonHandler(self):
    pyautogui.press('enter')

#!/usr/bin/env python
# -*- coding: cp1252 -*-
# Program Description: Runs the controller loop for the system
# Date Last Modified: December 05, 2016
# File name: controller.py

#Imports
import struct, time, math, csv
from PyQt4 import QtCore, QtGui, uic
from hardwareState import hardwareState

#-------------------------Constants----------------------------------#
#Controller proportional constant
#TODO test system to determine these heating constants
KP_HEAT = 0xFF
FAN_HEAT_SPEED = 0xFF
MOTOR_SPEED = 0xC0
ON = 0x01
OFF = 0x00

#Control Timing-------------#
#Control loop update period
CONTROL_PERIOD = 30
#Time to incubate
INCUBATION_TIME_SECONDS = 3600.0

#--------------------Commands--------------------#
#The following commands issue a status request in
#addition to their particular function
Sets motor duty cycle
Accepted values: 0-255
\[ D = \frac{\text{MOTOR\_DUTY\_SET\_VALUE}}{255} \]
MOTOR\_DUTY\_SET = 0x08

Sets fan duty cycle
Accepted values: 0-255
\[ D = \frac{\text{FAN\_DUTY\_SET\_VALUE}}{255} \]
FAN\_DUTY\_SET = 0x09

Turns fan on/off (Active low)
On = 0
Off = 1
FAN\_POWER\_SET = 0x0A

Sets heater duty cycle
Accepted values: 0-255
\[ D = \frac{\text{HEATER\_DUTY\_SET\_VALUE}}{255} \]
HEATER\_DUTY\_SET = 0x0B

Sets frequency of pwm pins
Accepted values: 11-15
Command values correspond to the following frequencies:
11 => 31.250kHz
12 => 3.906kHz
13 => 488Hz
14 => 122Hz
15 => 30.5Hz
FREQ\_SET = 0x0C

#----------------------------------------------------------------------------------#
#  Class Description: Runs control system, handles all error conditions, updates  #
#  GUI
# Last Edited: 12/4/2016
# Last Edited By: Pete Wirges
# Last Changes Made:
#  1.0.1: Moved class to its own file, added updateHandler
#  1.0.2: Added incubationTimer, finishIncubation, stopSystem
#  1.0.3: set updateHandler to check door and pressure
#         switches, implemented controller functionality inside
#         run function
#  1.0.4: Made control loop timer based, added startControlTimer,
#         stopControlTimer, systemHandler to handle user input on
#         which should be run, added signaling for buttons
#  1.1: Added save functionality, updated incubation configuration,
#        added safety catches, added system completion functionality
class controller(QtCore.QObject):
    #signal to send updated temperatures to gui
    tempUpdate = QtCore.pyqtSignal(float)
    #signals for messages to user
    doorSafetyWarning = QtCore.pyqtSignal()
    incubationFinishedMessage = QtCore.pyqtSignal()
    #System state update: 0 = Idle, 1 = Heating, 2 = Incubating, 3 = Complete
    systemUpdate = QtCore.pyqtSignal(int)
    #Tactile input updates
    upPressed = QtCore.pyqtSignal()
    downPressed = QtCore.pyqtSignal()
    backPressed = QtCore.pyqtSignal()
    selectPressed = QtCore.pyqtSignal()
    #Display timer signals
    startGuiTimer = QtCore.pyqtSignal(bool)  #bool =
    stopGuiTimer = QtCore.pyqtSignal()
    #Timer for updating hardware status/sending commands
    updateTimer = QtCore.QTimer()

    #Define format for writing to csv
    csv.register_dialect('bloodWarmerDialect',
        delimiter = '	',
        quoting = csv.QUOTE_MINIMAL)

def __init__(self):
    super(self.__class__, self).__init__()
    #Initialize hardware model
    self.arduino = hardwareState()
    #Initialize controller variables
    self._lastHeaterDutyByte = 0
    self._kp = KP_HEAT
    self._setTemp = 37.0
    self._tempAvg = 0.0
    #Initialize status flags
    self._running = False
    self._incubating = False
    self._ready = False
    #Initialize time tracking variables
    self._incStartTime = 0
    self._incTime = 0
    self._heatTime = 0
    self._heatStart = 0
    #Initialize save file
    self.save = None
    self.updateTimer.timeout.connect(self.runSystem,QtCore.Qt.QueuedConnection)
    self.startUpdateTimer()
Description: Runs system based on user input
Inputs: systemState - if the system should be running or not
        restart - If the system is resuming (0), or restarting (1)
Outputs: None

@QtCore.pyqtSlot(bool, bool)
def systemHandler(self, systemState, restart):
    # Stop control loop and sleep until last callback made
    self.stopUpdateTimer()
    time.sleep(1)
    # Configure display timer on restart
    self.startGuiTimer.emit(restart)
    # Configure heating time on restart
    if restart:
        self._heatStartTime = time.time() - self._heatTime
    # If returning from door safety fault, start system only if door is closed, otherwise repeat fault
    if systemState and self.arduino.doorSwitch:
        self.startSystem()
    elif not systemState and self.arduino.doorSwitch:
        self.stopSystem()
    else:
        self.stopSystem()
        self.doorSafetyWarning.emit()
    # Restart control loop timer
    self.startUpdateTimer()

Description: Enables logging of temperature over time
Inputs: None
Outputs: None

@QtCore.pyqtSlot()
def enableSaving(self):
    try:
        self.saveFile = open('/media/pi/USB/bloodwarmerData.csv', 'a')
    except:
        self.saveFile = None

Description: Configures and starts control timer
Inputs: None
Outputs: None

def startUpdateTimer(self):
    self.updateTimer.start(CONTROL_PERIOD)

Description: Stops control timer
Inputs: None
Outputs: None
def stopUpdateTimer(self):
    self.updateTimer.stop()

def updateHandler(self):
    # Set temperature to be controlled
    self._tempAvg = self.arduino.bagTempAvg
    self.tempUpdate.emit(self._tempAvg)

    # Send average temperature to gui
    self.tempUpdate.emit(self._tempAvg)

    # Call tactile input event handlers
    if self.arduino.upSwitch:
        self.upPressed.emit()
    if self.arduino.downSwitch:
        self.downPressed.emit()
    if self.arduino.backSwitch:
        self.backPressed.emit()
    if self.arduino.selectSwitch:
        self.selectPressed.emit()

    # Checks if door is open and sets safety warning if not
    if self._running and not self.arduino.doorSwitch:
        self.stopSystem()
        self.doorSafetyWarning.emit()

def startSystem(self):
    # Set control loop running flag
    self._running = 1

    # Save time and average temperature to flash drive if save button has been pressed
    if self.saveFile:
        try:
            self.saveFile = open('/media/pi/USB/bloodwarmerData.csv', 'a')
            dataWriter = csv.writer(self.saveFile, dialect = "bloodWarmerDialect")
            dataWriter.writerow(('Date/Time', 'Average Bag Temperature (C)'))
        except:
            pass

    # Initialize motor and fan
    self.sendCmd(bytearray([MOTOR_DUTY_SET, MOTOR_SPEED]))
    self.sendCmd(bytearray([FAN_POWER_SET, ON]))
    self.sendCmd(bytearray([FAN_DUTY_SET, FAN_HEAT_SPEED]))

    # Determine system state starting in, configure controller and display appropriately
    error = self._setTemp - self._tempAvg
    if error <= 0.5 and ~self._incubating:
        self._incubating = True
self._incStartTime = time.time()
self.startGuiTimer.emit(True)
self.systemUpdate.emit(2)
elif self._incubating:
    self.systemUpdate.emit(2)
else:
    self.systemUpdate.emit(1)

---

Description: Sends command to stop control system hardware
Inputs: None
Outputs: Off commands to arduino

---

def stopSystem(self):
    #Set control loop running flag to 0
    self._running = 0
    #Stops motor, fan, and heater
    self.sendCmd(bytearray([MOTOR_DUTY_SET, OFF]))
    self.sendCmd(bytearray([FAN_POWER_SET, OFF]))
    self.sendCmd(bytearray([HEATER_DUTY_SET, OFF]))
    #Stops display timer and updates system state displayed
    self.stopGuiTimer.emit()
    self.systemUpdate.emit(0)
    #Closes save file if open
    try:
        self.saveFile.close()
    except:
        pass

---

Description: Updates the set reference temperature used in control loops
Inputs: newSetTemp - user specified set temp between 37.0-41.0C
Outputs: None

---

QtCore.pyqtSlot(float)
def updateSetTemp(self,newSetTemp):
    self._setTemp = newSetTemp

---

Description: Sends command to atmega controller, stops system on temp sensor fault
Inputs: cmd - command to send to arduino
Outputs: None

---

def sendCmd(self, cmd):
    #Send command
    update = self.arduino.sendCmd(cmd)
    #Handle model update
    if update == 1:
        self.updateHandler()
    #Stop system on temp sensor fault
    if update == 2:
        self.stopSystem()

---

Description: Control system run function, signaled by start button,
Inputs: None
Outputs: Commands to arduino
def runSystem(self):
    # Run control loop
    if self._running:
        # If save button has been pressed, save time and average temperature to file
        if self.saveFile:
            try:
                logTime = time.asctime(time.localtime(time.time()))
                dataWriter = csv.writer(self.saveFile, dialect = "bloodWarmerDialect")
                dataWriter.writerow((logTime, "%.2f" % self._tempAvg))
            except:
                pass

        # Calculate the time spent heating
        self._heatTime = time.time() - self._heatStartTime
        # Get error for controller
        error = self._setTemp - self._tempAvg
        # Floor negative error values to 0
        if error < 0:
            error = 0
        # Check and configure for system state (heating/incubation)
        # Start incubating
        if error <= 0.5 and not self._incubating:
            # Set incubation flag
            self._incubating = True
            # Get incubation time
            self._incStart = time.time() - self._incTime
            # Restart GUI timer and update system state to incubating
            self.startGuiTimer.emit(True)
            self.systemUpdate.emit(2)
            # Stop incubating
        elif error > 0.5 and self._incubating:
            # Set incubation flag
            self._incubating = False
            # Reset incubation time and switch system state to heating
            self._incStart = 0
            self._incTime = 0
            self.systemUpdate.emit(1)
        else:
            # Calculate heater duty signal to send based on constant
            duty = error * self._kp
            # Ceiling function for duty value
            if duty > 255:
                duty = 255
            # Convert duty value from float to byte
            dutyByte = struct.pack("B", duty)
            # Set duty cycle of heater
            if dutyByte is not self._lastHeaterDutyByte:
                self.sendCmd(bytearray([HEATER_DUTY_SET, dutyByte]))
                self._lastHeaterDutyByte = dutyByte
            # Update incubation time
            if self._incubating:
                self._incTime = time.time() - self._incStart
                # Check for incubation completion and signal to display
                if self._incTime >= INCUBATION_TIME_SECONDS and not self._ready:
                    self.systemUpdate.emit(3)
                    self.incubationFinishedMessage.emit()
self._ready = True
else:
    # Get hardware update only if controller not running
    self.sendCmd(bytearray([STATUS_REQUEST, 0x00]))

# Program Description: Communication to the Arduino
# Date Last Modified: December 05, 2016
# File name: arduinoComm.py
#
# import serial
import math
import time
import struct
import binascii
import logging
import RPi.GPIO as GPIO
from PyQt4 import QtCore

#----------------Constants-----------------------#
# Pi GPIO pin to reset Atmega
ATMEGA_RESET_PIN = 25

# Communication control characters
BEGIN = 0x02  # Start transmission
END = 0x03  # End transmission
ACK = 0x06  # Acknowledge packet
NACK = 0x15  # No acknowledge packet
ESC = 0x1B  # Escape character: indicates control characters within packet

#--------------------Commands--------------------#
STATUS_REQUEST = 0x07  # Returns status of all hardware

# The following commands issue a status request in addition to their particular function

Sets motor duty cycle
Accepted values: 0-255
D = MOTOR_DUTY_SET_VALUE/255

Sets fan duty cycle
Accepted values: 0-255
D = FAN_DUTY_SET_VALUE/255

Turns fan on/off (Active low)
On = 0  
Off = 1  

FAN_POWER_SET = 0xA  

Sets heater duty cycle  
Accepted values: 0-255  
D = HEATER_DUTY_SET_VALUE/255  

HEATER_DUTY_SET = 0xB  

Sets frequency of pwm pins  
Accepted values: 11-15  
Command values correspond to the following frequencies:  
11 => 31.250kHz  
12 => 3.906kHz  
13 => 488Hz  
14 => 122Hz  
15 => 30.5Hz  

FREQ_SET = 0xC

#------------------------------------------------#  
#-----------Serial port config values------------#  
SERIALPORT = '/dev/serial0'  
BAUD_RATE = 19200

#debug file  
LOGFILE = '/home/pi/Downloads/comm.log'


Class Description: For sending commands/receiving status from AtMega  
Last Edited: 12/4/2016  
Last Edited By: Pete Wirges  
Changelog: -1.0.0: Established protocol to send commands to arduino and receive status updates and store them as properties  
-1.0.1: Added threading support, removed properties, added stopOutput function to reset hardware, improved documentation, added handling for case when checksum is invalid  
-1.0.1: Removed remnants of unused properties, adjusted packet size for fan on/off control  
-1.0.1: Updated to include fan power control/status  
-1.0.2: Added reset of atmega at startup, removed receive delay to tune comm time from 150ms to 26ms

Hardware state values:  
Command(See list of available commands at top of file):  
  cmdType  
  cmdValue

Tactile Input (1 = ON, 0 = OFF):  
  upSwitch  
  downSwitch
selectSwitch
backSwitch

Safety Switches (1 = ON, 0 = OFF):
pressureSwitch1
pressureSwitch2
doorSwitch

Temperature Sensors (in degrees Celcius):
bag1TempC
gag2TempC
inletTempC
outletTempC

Output States (indicated by corresponding command value (see above)):
motorState
fanState
heaterState
pwmFrequency

class arduinoComm(QtCore.QObject):
    'Class for sending commands to and receiving status from arduino'

    #To send hardware status to another thread
    hardwareStatusUpdate = QtCore.pyqtSignal(bytearray)

    def __init__(self, parent = None, runCmd = 0, cmdType = STATUS_REQUEST, cmdValue = 0x00, checksum = 0x00):
        super(self.__class__, self).__init__(parent)
        #TODO Add code here to hold atmega reset pin high for 2s before attempting serial communication
        #Initialize serial port
        self.ser=serial.Serial(port=SERIALPORT, baudrate=BAUD_RATE, bytesize=8, parity = 'N', stopbits = 1)
        self.ser.close()
        self.ser.open()
        #Configure Atmega reset pin
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(ATMEGA_RESET_PIN, GPIO.OUT)
        #Initialize debug logger
        self.logger = logging.getLogger('arduinoComm')
        hdlr = logging.FileHandler(LOGFILE)
        formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
        hdlr.setFormatter(formatter)
        self.logger.addHandler(hdlr)
        self.logger.setLevel(logging.DEBUG)
        self.logger.debug('initializing communications')
        #Initialize hardware properties
        self._runCmd = runCmd
        self._cmdType = cmdType
        self._cmdValue = cmdValue
        self._checksum = checksum
        #Initialize outputs as zero
        self.stopOutput()
def sendCmd(self, cmd):
    self.logger.debug('sending command')
    # Encode command for control characters
    cmd = self.encodeCmd(cmd)
    # Calculate checksum for packet
    crc = self.CRC8(cmd)
    # Clear serial buffers
    self.ser.flushInput()
    self.ser.flushOutput()
    # Frame packet and send
    cmd = bytearray([BEGIN]) + cmd + bytearray([crc]) + bytearray([END])
    self.ser.write(cmd)
    # Receive response
    resp = self.readRsp()
    return self.processRsp(resp)

def encodeCmd(self, cmd):
    encodedCmd = bytearray([])
    # Checks each byte for control character,
    # If found, sets bit 7 high and inserts ESC before it
    for i in cmd:
        if (i == BEGIN) or (i == END) or (i == ESC):
            encodedCmd = encodedCmd + bytearray([ESC]) + bytearray([(i ^ (1 << 7))])
        else:
            encodedCmd = encodedCmd + bytearray([i])
    return encodedCmd

def CRC8(self, data):
    crc = 0x00
    for e in data:
        for i in range(8):
            s = (crc ^ e) & 0x01
            crc >>= 1
            if (s):
                crc ^= 0x8C
            e >>= 1
    return crc
def stopOutput(self):
    GPIO.output(ATMEGA_RESET_PIN, GPIO.HIGH)
    time.sleep(2)
    GPIO.output(ATMEGA_RESET_PIN, GPIO.LOW)
    time.sleep(2)
    self.sendCmd(bytearray([MOTOR_DUTY_SET, 0x00]))
    self.sendCmd(bytearray([FAN_DUTY_SET, 0x00]))
    self.sendCmd(bytearray([FAN_POWER_SET, 0x00]))
    self.sendCmd(bytearray([HEATER_DUTY_SET, 0x00]))
    self.sendCmd(bytearray([FREQ_SET, 0x11]))

    def readRsp(self):
        resp = bytearray()
        trans = False
        # Wait for response
        self.logger.debug('Waiting for response')
        while self.ser.inWaiting > 0:
            c = self.ser.read(1)
            # Wait for BEGIN to read response, stop reading after END
            if c == bytearray([BEGIN]):
                trans = True
            elif trans:
                if c == bytearray([END]):
                    break
                else:
                    resp += c
            return resp

    def processRsp(self, resp):
        # Check if packet is valid and if so extract it
        resp = self.extractPacket(resp)
        if (resp != None):
            # Decode packet if valid
            if (len(resp) > 0):
                resp = self.decodeCtrlChar(resp)
                # Update status
                if resp[0] == NACK:
                    self.logger.debug('Command not acknowledged')
                    return bytearray([NACK])
                else:
                    return resp
            else:
                return bytearray([NACK])
        else:
            return bytearray([NACK])
Description: Extracts valid packet from received transmission
Inputs: Received packet with BEGIN and END removed
Outputs: If checksum is valid, returns packet with checksum removed. Otherwise returns None.

```python
def extractPacket(self, resp):
    self.logger.debug('Checking Packet')
    lenresp = len(resp)
    status = bytearray()
    # Get received checksum
    self._checksum = resp[lenresp-1]
    # Extract packet from received transmission
    for i in range(lenresp-1):
        status += bytearray([resp[i]])
    # Calculate checksum and compare it with received
    if (self._checksum == self.CRC8(status)):
        self.logger.debug('Valid status')
        return status
    else:
        self.logger.debug('Invalid Checksum')
        return None
```

Description: Decode control characters from received packet
Inputs: Valid packet with frame completely removed
Outputs: Received packet decoded for control characters

```python
def decodeCtrlChar(self, status):
    self.logger.debug('Decoding Control Characters')
    decodedStatus = bytearray([])
    # If ESC found, ignore and decode next character by setting bit 8 LOW
    for i in range(len(status)):
        if (status[i] == ESC):
            if ((i > 0) and (status[i-1] == ESC)):
                decodedStatus += bytearray([status[i]])
            elif (i < len(status)-1):
                status[i+1] = status[i+1] & 0x7F
            else:
                decodedStatus += bytearray([status[i]])
    return decodedStatus
```
# Date Last Modified: December 05, 2016
# File name: hardware.py
#
#----------------------------------------------------------------------------------#

# imports
import math
import struct
from PyQt4 import QtCore
from arduinoComm import arduinoComm

NACK = 0x15  # No acknowledge packet

#-------------------Commands-------------------#
STATUS_REQUEST = 0x07  # Returns status of all hardware

# The following commands issue a status request in
# addition to their particular function

"""
Sets motor duty cycle
Accepted values: 0-255
D = MOTOR_DUTY_SET_VALUE/255
"""
MOTOR_DUTY_SET = 0x08

"""
Sets fan duty cycle
Accepted values: 0-255
D = FAN_DUTY_SET_VALUE/255
"""
FAN_DUTY_SET = 0x09

"""
Turns fan on/off (Active low)
On = 0
Off = 1
"""
FAN_POWER_SET = 0x0A

"""
Sets heater duty cycle
Accepted values: 0-255
D = HEATER_DUTY_SET_VALUE/255
"""
HEATER_DUTY_SET = 0x0B

"""
Sets frequency of pwm pins
Accepted values: 11-15
Command values correspond to the following frequencies:
11 => 31.250kHz
12 => 3.906kHz
13 => 488Hz
14 => 122Hz
15 => 30.5Hz
"""
FREQ_SET = 0x0C
#--------Temp Sensor calibration offsets---------#
BAG11_CAL = -0.1
BAG12_CAL = -0.2
BAG21_CAL = -0.1
BAG22_CAL = -0.2
#------------------------------------------------#

'''----------------------------------------------------------------------------
Class Description: Model of blood warmer hardware, provides status of IO and
interrupts controller/gui on update
Last Edited: 11/28/2016
Last Edited By: Pete Wirges
Changelog: -1.0.0: Created properties, functions to send command to comm link,
parse updates from comm link, push temperatures to gui
    -1.0.1: Added temperature averaging, modified signal emitted to notify
     controller rather than update gui, commented file
    -1.0.2: Updated to include fan power control/status
    -1.0.3: Updated timer callback function (sendCmd) to send all commands,
        added setCmd slot that receives commands from controller thread
        and sets the next cmd to send. This way there won't be multiple
        sendCmd requests and all commands will execute at a fixed interval,
        fixed bug in commands
    -1.0.4: Reconfigured to run in controller
----------------------------------------------------------------------------`````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````
Description: Timer callback function, sends command to comm link, defaults to STATUS_REQUEST after transmission
Inputs: cmdType, cmdValue
Outputs: None

def sendCmd(self, cmd):
    response = self._serial.sendCmd(cmd)
    return self.parseStatus(response, cmd)

Description: Parses received hardware status packet into hardware model, notifies controller about update
Inputs: Decoded hardware status packet
Outputs: emits update

def parseStatus(self, status, cmd):
    if status[0] == NACK:
        return 0;
    else:
        # Update user input
        self._upSwitch = status[1]
        self._downSwitch = status[2]
        self._selectSwitch = status[3]
        self._backSwitch = status[4]
        # Update safety switches
        self._pressureSwitch1 = status[5]
        self._pressureSwitch2 = status[6]
        self._doorSwitch = status[7]
        # Unpack and update temperatures
        newBag11TempC, = struct.unpack("f", bytearray([status[8], status[9], status[10], status[11]]))
        newBag12TempC, = struct.unpack("f", bytearray([status[12], status[13], status[14], status[15]]))
        newBag21TempC, = struct.unpack("f", bytearray([status[16], status[17], status[18], status[19]]))
        newBag22TempC, = struct.unpack("f", bytearray([status[20], status[21], status[22], status[23]]))
        if newBag11TempC < 0 or newBag12TempC < 0 or newBag21TempC < 0 or newBag22TempC < 0:
            self.systemError.emit("System Failure")
            return 2
        # Sensor Calibration
        newBag11TempC -= BAG11_CAL
        newBag12TempC -= BAG12_CAL
        newBag21TempC -= BAG21_CAL
        newBag22TempC -= BAG22_CAL
        # Get average reading for each bag
        newBag11TempC = (newBag11TempC + newBag12TempC) / 2
        newBag21TempC = (newBag21TempC + newBag22TempC) / 2
        # Get average of last 10 readings
        self._bag1TempAvg = self.avgTemp(self._bag1Readings, newBag11TempC)
        self._bag2TempAvg = self.avgTemp(self._bag2Readings, newBag21TempC)
        # Ensure both bags are in system; if one is not included, take the lower temperature
        if self._bag1TempC - self._bag2TempC > 1:
            self._bagTempAvg = self._bag2TempC
        if self._bag2TempC - self._bag1TempC > 1:
            self._bagTempAvg = self._bag1TempC
        else:
            self._bagTempAvg = (self._bag1TempC + self._bag2TempC) / 2
# Update output status
self._motorDutyState = status[24]
self._fanPowerState = status[25]
self._fanDutyState = status[26]
self._heaterDutyState = status[27]
self._pwmFrequency = status[28]
return 1;

---

Description: Calculates and returns a running average of the last ten temperature readings
Inputs: tempReadings = List of last 10 readings, newTemp = Latest reading
Outputs: avgTemp

```python
def avgTemp(self, tempReadings, newTemp):
    tempReadings.insert(0,newTemp)
    readings = len(tempReadings)
    avgTemp = 0
    if (readings > 10): 
        tempReadings.remove(tempReadings[readings-1])
        readings -= 1
    for temp in tempReadings:
        avgTemp += temp
    avgTemp /= readings
    return avgTemp
```

---

##----------------------Hardware properties for state access------------------------

@property
def cmdType(self):
    return self._cmdType

@cmdType.setter
def cmdType(self, value):
    self._cmdType = value

@property
def cmdValue(self):
    return self._cmdValue

@cmdValue.setter
def cmdValue(self, value):
    self._cmdValue = value

@property
def checksum(self):
    return self._checksum

@checksum.setter
def checksum(self, value):
    self._checksum = value

@property
def upSwitch(self):
    return self._upSwitch

@upSwitch.setter
def upSwitch(self, value):
    self._upSwitch = value
@property
def downSwitch(self):
    return self._downSwitch

@downSwitch.setter
def downSwitch(self, value):
    self._downSwitch = value

@property
def selectSwitch(self):
    return self._selectSwitch

@selectSwitch.setter
def selectSwitch(self, value):
    self._selectSwitch = value

@property
def backSwitch(self):
    return self._backSwitch

@backSwitch.setter
def backSwitch(self, value):
    self._backSwitch = value

@property
def pressureSwitch1(self):
    return self._pressureSwitch1

@pressureSwitch1.setter
def pressureSwitch1(self, value):
    self._pressureSwitch1 = value

@property
def pressureSwitch2(self):
    return self._pressureSwitch2

@pressureSwitch2.setter
def pressureSwitch2(self, value):
    self._pressureSwitch2 = value

@property
def doorSwitch(self):
    return self._doorSwitch

@doorSwitch.setter
def doorSwitch(self, value):
    self._doorSwitch = value

@property
def bag1TempC(self):
    return self._bag1TempC

@bag1TempC.setter
def bag1TempC(self, value):
    self._bag1TempC = value

@property
def bag2TempC(self):
    return self._bag2TempC
@bag2TempC.setter
def bag2TempC(self, value):
    self._bag2TempC = value

@property
def bagTempAvg(self):
    return self._bagTempAvg

@bagTempAvg.setter
def bagTempAvg(self, value):
    self._bagTempAvg = value

@property
def pwmFrequency(self):
    return self._pwmFrequency

@pwmFrequency.setter
def pwmFrequency(self, value):
    self._pwmFrequency = value

@property
def motorDutyState(self):
    return self._motorDutyState

@motorDutyState.setter
def motorDutyState(self, value):
    self._motorDutyState = value

@property
def fanDutyState(self):
    return self._fanDutyState

@fanDutyState.setter
def fanDutyState(self, value):
    self._fanDutyState = value

@property
def heaterDutyState(self):
    return self._heaterDutyState

@heaterDutyState.setter
def heaterDutyState(self, value):
    self._heaterDutyState = value

# WARNING POPUP FILE
#-------------------------------------------#

# LOAD GUI FILE
#-------------------------------------------#
qtMyPopUpFile = "/home/pi/Documents/BloodWarmer/warningPopUp.ui"
Ui_PopUpWindow, QtSubClass = uic.loadUiType(qtMyPopUpFile)
class warningPopup(QtGui.QWidget, Ui_PopUpWindow):

    runSystem = QtCore.pyqtSignal(bool, bool)

def __init__(self):
    QtGui.QWidget.__init__(self)
    Ui_PopUpWindow.__init__(self)
    self.setupUi(self)

    # User Interface Property changes
    # ---------------------------------------------------#

    # Call appropriate function on user actions
    # ---------------------------------------------------#
    # Connects to startSystem() function
    #self.continueButton.clicked.connect(self.close)
    #self.restartButton.clicked.connect(self.close)

    @QtCore.pyqtSlot()
    def setWarning(self):
        # Freezes the Main Window till a response is made by the user for MyPopup()
        self.setWindowModality(QtCore.Qt.ApplicationModal)
        self.show()

    @QtCore.pyqtSlot()
    def chooseResume(self):
        self.runSystem.emit(True, False)
        self.close()

    @QtCore.pyqtSlot()
    def chooseRestart(self):
        self.runSystem.emit(True, True)
        self.close()

# INCLUDES
#-----------------------------------------------------------#

from PyQt4 import QtCore, QtGui, uic
#-----------------------------------------------------------#

# LOAD GUI FILE
#-----------------------------------------------------------#
class errorPopup(QtGui.QWidget, Ui_PopUpWindow):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        Ui_PopUpWindow.__init__(self)
        self.setupUi(self)
        self.errorButton.clicked.connect(self.close)

    @QtCore.pyqtSlot(str)
    def setError(self, errorText):
        self.setText(errorText)
        # Freezes the Main Window till a response is made by the user for MyPopup()
        self.setWindowModality(QtCore.Qt.ApplicationModal)
        self.show()
Appendix C: AtMega Control Board Code

bwHardware.h

```c
#ifndef bwHardware_h
#define bwHardware_h

#include <Wire.h>
#include <Arduino.h>
#include <p1Comm.h>
#include <inputOutput.h>
#include <safetySensors.h>
#include <tempSensors.h>
#include <outputControl.h>

const byte PACKET_SIZE = 29;

//Class: BWHardware
//Description: Wrapper class for interfacing with hardware and communicating back to pi

class bwHardware {

public:

	/////////Interface/////////

bwHardware(); //constructor
void initialize(const HardwareSerial *s, const int baudRate);//initializes hardware and serial port
void respond(void); // Receives/executes commands from pi and returns status

private:

	/////////Private variables/////////

//Hardware instances
p1Comm pi; //Serial communication with raspberry pi
inputOutput ui; //UI tactile switches
safetySensors safety; //Sensors involved with safety conditions
tempSensors tempArray; //Array of temperature sensors
outputControl output; //Motor, fan, and heater control

	/////////Private Functions/////////
byte* getStatus(); //Gets status of inputs/outputs

};

#endif
```
#include <bwHardware.h>

// Constructor
bwHardware::bwHardware()
{
}

// Function: initialize
// Description: Initializes all blood warmer hardware
// Inputs: None
// Output: None
void bwHardware::initialize(const HardwareSerial *s, const int baudRate)
{
    pi.initialize(s, baudRate);
    ui.initialize();
    output.initialize();
    safety.initialize();
    tempArray.initialize();
}

// Function: getStatus
// Description: Fetches state of all hardware and stores it in buffer
// Inputs: None
// Output: None
byte* bwHardware::getStatus(void)
{
    static byte status[PACKET_SIZE] = {0};
    // Get status of subsystems
    byte* uiStatus = ui.getStatus();
    byte* safetyStatus = safety.getStatus();
    byte* tempReadings = tempArray.readAllSensors();
    byte* outputStatus = output.getStatus();

    // Add individual statuses together
    memcpy(&status[], uiStatus, 4);
    memcpy(&status[4], safetyStatus, 3);
    memcpy(&status[7], tempReadings, 16);
    memcpy(&status[24], outputStatus, 5);
    return status;
}

// Function: respond
// Description: Waits for and executes requests sent by Raspberry Pi on serial line
// Inputs: None
// Output: None
void bwHardware::respond(void)
{
    byte* hardwareStatus;
    // Received command variables
    byte cmdType;
    byte cmdValue;
    // See if a valid transmission has been received, if not send nack
    if (pi.cmdInWaiting())
    {
        if (pi.receive())
        {
            // Get command from transmission and execute
            cmdType = pi.getCmdType();
            cmdValue = pi.getCmdValue();
            if (cmdType != STATUS_REQUEST) output.execute(cmdType, cmdValue);
            // Get status of hardware and send reply
            hardwareStatus = getStatus();
            pi.sendAck(hardwareStatus);
        }
    }
    else pi.sendNack();
}
piComm.h

```c
#ifndef piComm_h
#define piComm_h

#include "Archino.h">

//==================================================================================================

//==================================================================================================

const byte BEGIN = 0x02; //Begin transmission
const byte END = 0x03; //End Transmission
const byte ACK = 0x06; //Acknowledgement
const byte NACK = 0x15; //No acknowledge
const byte ESC = 0x12; //Escape character

//==================================================================================================

const byte MAX_RECEIPTION_LENGTH = 6; //Maximum length of received command
const byte STATUS_LENGTH = 29; //Maximum length of hardware status
const byte ENCODED_STATUS_LENGTH = 43; //Maximum length of encoded status

//==================================================================================================

class piComm {
  public:
    ///////////interface/////////

    piComm(void); //Constructor

    void initialize(const HardwareSerial *, const int baudRate); //Initializes serial port connection with pi
    bool cmdIsWaiting(); //Indicates whether a command has been sent on the serial port
    byte getCmdType(); //Get the command type
    byte getChValue(); //Get the value for the command
    bool receive(void); //Receives incoming command
    void sendNack(); //Sends nack
    void sendAck(const byte * hardwareStatus); //Encodes and sends status of hardware

  private:
    HardwareSerial *serial; //Serial port connected to pi

    ///////////Private variables/////////

    //For receiving and executing commands
    byte commandType; //Can be status request, motor set, fan set, heater set, pwm set
    byte commandValue; //Value for set functions: duty cycle numerator for motor/fan/heater, frequency divisor for pwm

    //For transmission
    byte statusLength; //Length of encoded status to send, varies depending on control characters present in message

    ///////////Private Functions/////////

    byte CRC8(const byte * data, const byte len); //Calculate CRC8 Dallas/Maxim checksum
    byte* encodeCtrlChar(const byte* hardwareStatus); //Encodes status for control characters
    void decodeCmd(byte* encodedCmd, const byte cmdLength); //Decodes for control characters in received packet

};
```

113
#include <picComm.h>

piComm::piComm()
{
    serial = NULL;
}

void piComm::initialize(const HardwareSerial *s, const int baudRate)
{
    serial = s;
    serial->begin(baudRate);
}

//Function: CRC8
//Description: Compute 8 bit CRC checksum for use in serial transmission
//Inputs:
//  -data - byte array containing data to be checked
//  -len - the length of the data to be checked in bytes
//Output: 8-bit checksum
byte piComm::CRC8(const byte *data, const byte len)
{
    byte crc = 0x00;
    for (byte i = 0; i < len; i++)
    {
        byte c = data[i];
        for (byte j = 0; j < 8; j++)
        {
            byte sum = (crc ^ c) & 0x01;
            crc >>= 1;
            if (sum)
            {
                crc = crc ^ 0x8C;
            }
            c >>= 1;
        }
        return crc;
    }
}

//Function: getCmdType
//Description: Getter for commandType
//Inputs: None
//Outputs: commandType
byte piComm::getCmdType()
{
    return commandType;
}

//Function: setCmdType
//Description: Setter for commandType
//Inputs: commandType
//Outputs: None
byte piComm::setCmdType(byte cmd)
{
    commandType = cmd;
    return commandType;
}

//Function: getCmdValue
//Description: Getter for commandValue
//Inputs: None
//Outputs: commandValue
byte piComm::getCmdValue()
{
    return commandValue;
}

//Function: setCmdValue
//Description: Setter for commandValue
//Inputs: commandValue
//Outputs: None
byte piComm::setCmdValue(byte cmd)
{
    commandValue = cmd;
    return commandValue;
}

//Function: encodeCtrlChar
//Description: Encode control characters for use in transmission. If found, sets bit 7 of byte high and inserts escape character before.
//Inputs: -vector<Byte> &hardwareStatus: unencoded hardware status
//Output: -bytes<encoded status: status encoded for control characters in an array for transmission
byte piComm::encodeCtrlChar(const byte *hardwareStatus)
{
    static byte encodedStatus[ENCOD_STATUS_LENGTH] = {0};
    //Iterate through status buffer
    byte j = 0;
    for (byte i = 0; i < STATUS_LENGTH; i++)
    {
        if (hardwareStatus[i] == ESC) { hardwareStatus[i] = ESC; //HardwareStatus[i] = ESC; }
    }
}
void piComm::decodeCmd(byte *encodedCmd, const byte cmdLength) {
    byte j = 0;
    for (byte i = 0; i < cmdLength; i++) {
        if (encodedCmd[i] == ESC) {
            if (i > 0 && encodedCmd[i-1] == ESC) {
                encodedCmd[i] = encodedCmd[i+1];
            } else {
                encodedCmd[i] = encodedCmd[i+1] & 0x7F;
            }
        }
        decodedCmd[j] = encodedCmd[i];
        j++;
    }
    commandType = decodedCmd[0];
    commandValue = decodedCmd[1];
    }

// Function: Receive
// Description: Receive and verify valid transmission
// Inputs: None
// Outputs: bool: True if successful valid reception
bool piComm::receive(void) {
    byte bytesReadToRead; //Bytes waiting on the serial port
    byte commandLength; //Length of received packet
    byte checkSum; //CRC3 Checksum
    byte receiveBuffer[MAX_RECEPTION_LENGTH]; //Buffer to hold received command
    bool received = false;
    delay();
    bytesReadToRead = (byte) serial->available();
    if (serial->peek() == ESC) {
        serial->readBytes(receiveBuffer, bytesReadToRead);
        checkSum = receiveBuffer[bytesToRead-1];
        commandLength = byteToRead-1;
        if (checkSum == CRC3(receiveBuffer, commandLength)) \n        if (receiveBuffer[bytesToRead-1] == ESC) {
            received = true;
        } else received = false;
    }
    return received;
}
bool piComm::cmdInWaiting()
{
    if (serial->available() > 0) return true;
    else return false;
}

//Function: sendNack
//Description: Send no acknowledge response
//Inputs: None
//Output: None
void piComm::sendNack()
{
    //Send NACK to Pi if checksum does not match
    byte checksum = CRC8(NACK, 1);
    serial->write(BEGIN);
    serial->write(NACK);
    serial->write(checksum);
    serial->write(END);
}

//Function: sendAck
//Description: Send acknowledge response, includes hardware status and executed command
//Inputs: vector<byte> hardwareStatus; hardware status not yet encoded for transmission
//Output: None
void piComm::sendAck(const byte* hardwareStatus)
{
    //Encode status for control characters
    byte* encodedStatus = encodeCtrlChar(hardwareStatus);
    //Calculate checksum of encoded status
    checksum = CRC8(encodedStatus, statusLength);
    //Transmit response to Pi
    serial->write(BEGIN);
    serial->write(encodedStatus, statusLength);
    serial->write(checksum);
    serial->write(END);
}
```c
#include <Arduino.h>

const byte MOTOR_PIN = 11;
const byte FAN_PWM_PIN = 9;
const byte FAN_POWER_PIN = 7;
const byte HEATER_PIN = 8;

const byte STATUS_REQUEST = 0x07; // Return status
const byte MOTOR_DUTY_SET = 0x08; // Turn motor on/off, request status
const byte FAN_DUTY_SET = 0x09; // Set fan pwm, request status
const byte FAN_POWER_SET = 0x0A; // Turn power of fan on/off
const byte HEATER_DUTY_SET = 0x0B; // Set heater pwm, request status
const byte FREQ_SET = 0x0C; // Adjust frequency

const int PWM_FREQUENCY_DIVISOR FAN = 6;

class outputControl {
  public:
    // Interface
    outputControl(); // constructor
    void initialize(); // Sets pwm frequencies and initializes outputs to zero
    void execute(const byte cmdType, const byte cmdValue); // Executes a given command
    
  private:
    // Private Variables
    byte PWM_frequency; // above divisor coded as byte for communication
    byte motorDutyState; // Motor duty cycle: (0-255)/255
    byte fanDutyState; // Fan power On/Off
    byte fanDutyState; // Fan duty cycle: (0-255)/255
    byte heaterDutyState; // Heater duty cycle: (0-255)/255
    
    // Private functions
    
    void setPWMFrequency(const byte &pin, const int &divisor); // Set frequency of pwm pin given divisor
};
```
```cpp
#include <outputControl.h>

outputControl::outputControl() {}

// Function: initialize
// Description: Initialize output pins
// Inputs: None
// Output: None

void outputControl::initialize() {
    // Set pwm frequency for fan and heating element
    int pwmFrequencyDivisor = 1024;
    pinMode(FAN_POWER_PIN, OUTPUT);
    pwmFrequency = setPwmFrequency(HEATER_PIN, pwmFrequencyDivisor);
    setPwmFrequency(FAN_PWM_PIN, PWM_FREQUENCY_DIVISOR_FAN);
    // Set initial states of actuators
    motorDutyState = 0x00;
    fanDutyState = 0x00;
    heaterDutyState = 0x00;
    fanPowerState = 0x00;
    analogWrite(MOTOR_PIN, motorDutyState);
    digitalWrite(FAN_POWER_PIN, fanPowerState);
    analogWrite(FAN_PWM_PIN, fanDutyState);
    analogWrite(HEATER_PIN, heaterDutyState);
}

// Function: setPwmFrequency
// Source: Arduino Playground
// Description:
// Adjusts the frequency of pwm pins
// Inputs:
// - pin: The pin whose frequency is to be changed
// Output:
// - divisor - number to divide base frequency by
// Notes:
// The resulting frequency is equal to the base frequency divided by the given divisor:
// - Base frequencies:
//   - The base frequency for pins 3, 9, 10, and 11 is 31250 Hz.
//   - The base frequency for pins 5 and 6 is 62500 Hz.
// - Divisors:
//   - The divisors available on pins 5, 6, 9 and 10 are: 1, 5, 64,
//   - 256, and 1024.
// PWM frequencies are tied together in pairs of pins. If one in a
// pair is changed, the other is also changed to match:
// - Pins 5 and 6 are paired on timer0
// - Pins 9 and 10 are paired on timer1
// - Pins 3 and 11 are paired on timer2
// Note that this function will have side effects on anything else
// that uses timers:
// - Changes on pins 3, 5, 6, or 11 may cause the delay() and
//   millis() functions to stop working. Other timing-related
//   functions may also be affected.
// - Changes on pins 9 or 10 will cause the Servo library to function
//   incorrectly.

byte outputControl::setPwmFrequency(const byte pin, const int divisor) {
    byte mode;
    if (pin == 5 || pin == 6 || pin == 9 || pin == 10) {
        switch(divisor) {
            case 1: mode = 0x01; break;
```
```c
case 6: mode = 0x02; break;
case 64: mode = 0x03; break;
case 256: mode = 0x04; break;
case 1024: mode = 0x05; break;
default: return;
}

if(pin == 5 || pin == 6) {
    TCCR0B = TCCR0B & 0x11111000 | mode;
} else {
    TCCR1B = TCCR1B & 0b11111000 | mode;
}
return mode;
}
else if(pin == 3 || pin == 11) {
    switch(divisor) {
    case 1: mode = 0x01; break;
case 8: mode = 0x02; break;
case 32: mode = 0x03; break;
case 128: mode = 0x04; break;
case 256: mode = 0x05; break;
case 1024: mode = 0x07; break;
default: return;
    }
    TCCR2B = TCCR2B & 0b11111000 | mode;
    return mode18;
}
}

//Function: Execute
//Inputs: byte cmdType: Command type. Can be a status request, motor set, fan set, heater set, frequency set
//byte cmdValue: Command value, dependent on command type
//Output: None
void outputControl::execute(const byte cmdType, const byte cmdValue){
    int pwmFrequencyDivisor = 0;
    switch(cmdType) {
    case MOTOR_DUTY_SET: {
        motorDutyState = cmdValue;
        analogWrite(MOTOR_PIN, motorDutyState);
        return;
    }
    case FAN_POWER_SET: {
        fanPowerState = cmdValue;
        digitalWrite(FAN_POWER_PIN, cmdValue);
        return;
    }
    case FAN_DUTY_SET: {
        fanDutyState = cmdValue;
        analogWrite(FAN_PWM_PIN, fanDutyState);
        return;
    }
    case HEATER_DUTY_SET: {
        heaterDutyState = cmdValue;
        analogWrite(HEATER_PIN, heaterDutyState);
        return;
    }
    case FREQ_SET: {
        switch(cmdValue) {
        case 0x00: pwmFrequencyDivisor = 1; break;
        case 0x01: pwmFrequencyDivisor = 1; break;
        case 0x02: pwmFrequencyDivisor = 64; break;
        ```
case 0x0E: pwmFrequencyDivisor = 256; break;
case 0x0F: pwmFrequencyDivisor = 1024; break;
}
pwmFrequency = setPwmFrequency(HEATER_PIN, pwmFrequencyDivisor);
return;

// Function: getStatus
// Description: Gets status of output pins
// Inputs: None
// Output: vector< byte > status: [motor state, fan state, heater state, pwm frequency]

static byte status[5] = {0};
status[0] = motorDutyState;
status[1] = fanPowerState;
status[2] = fanDutyState;
status[3] = heaterDutyState;
status[4] = pwmFrequency;
return status;
```cpp
#include <Arduino.h>

const byte PRESSURE_SWITCH_1 = 14;
const byte PRESSURE_SWITCH_2 = 15;
const byte DOOR_SWITCH = 0;

const int DEBOUNCE_DELAY1 = 50;
const int DEBOUNCE_DELAY2 = 50;

class safetySensors {
  public:
    safetySensors(void); //constructor
    void initialize(void); //Initializes safety switches as inputs
    byte* getStatus(void); //Gets current values of safety switches
  private:
    //state variables for safety switches
    byte pressureSwitch1;
    byte pressureSwitch2;
    byte doorSwitch;
};

#endif
```
```cpp
#include <safetySensors.h>

safetySensors::safetySensors(){}

//Function: Initialize
//Description: Initialize safety switches as inputs
//Inputs: None
//Output: None

void safetySensors::initialize(){
    pinMode(PRESSURE_SWITCH_1, INPUT);
    pinMode(PRESSURE_SWITCH_2, INPUT);
    pinMode(Door_SWITCH, INPUT);
}

//Function: get status
//Description: Get status of safety switches.
//Inputs: None
//Output: vector<byte> status: [pressureSwitch1, pressureSwitch2, doorSwitch]

byte* safetySensors::getStatus(){
    static byte status[3] = {0};
    if (analogRead(PRESSURE_SWITCH_1) > 500) pressureSwitch1 = 0x01;
    else pressureSwitch1 = 0x00;
    if (analogRead(PRESSURE_SWITCH_2) > 500) pressureSwitch2 = 0x01;
    else pressureSwitch2 = 0x00;
    doorSwitch = digitalRead(DOOR SWITCH);
    status[0] = pressureSwitch1;
    status[1] = pressureSwitch2;
    status[2] = doorSwitch;
    return status;
}
```
tempSensors.h

```c
#ifndef tempSensors_h
#define tempSensors_h

#include <Adafruit_MCP9808.h>
#include <Arduino.h>

#define NAME

// I2C addresses of temperature sensors
const byte BAG_11_ADDRESS = 0x19;
const byte BAG_12_ADDRESS = 0x1A;
const byte BAG_21_ADDRESS = 0x1C;
const byte BAG_22_ADDRESS = 0x1D;

// Structure for allowing floating point to be broken into its 4 representative bytes
typedef union {
    float floatingPoint;
    byte binary[4];
} BinaryFloat;

class tempSensors {
    // Interface
    public:
    tempSensors(void);
    byte temp Sensors(void);
    byte* readAll Sensors(void);
    private:
    // Instance of each temperature sensor
    Adafruit_MCP9808 bag11Sensor;
    Adafruit_MCP9808 bag12Sensor;
    Adafruit_MCP9808 bag21Sensor;
    Adafruit_MCP9808 bag22Sensor;
    // State variable for temperature sensor readings in Celsius
    BinaryFloat bag11TempC;
    BinaryFloat bag12TempC;
    BinaryFloat bag21TempC;
    BinaryFloat bag22TempC;
    byte sensor11status;
    byte sensor12status;
    byte sensor21status;
    byte sensor22status;
    BinaryFloat readTempC(const Adafruit_MCP9808 &sensor);
);
```

123
```cpp
#include <tempSensors.h>

tempSensors::tempSensors(){
  // Find temperature sensors or throw exception (Make sure sensor physical address pins are wired correctly)
  byte sensor11Status = 0x00;
  byte sensor12Status = 0x00;
  if (bag1Sensor.begin(BAG11_ADDRESS)) {
    sensor11Status = 0x01;
  } else sensor11Status = 0x00;
  if (bag2Sensor.begin(BAG12_ADDRESS)) {
    bag1Sensor.shutdown_wake(0);
    sensor12Status = 0x01;
  } else sensor12Status = 0x00;
  if (bag1Sensor.begin(BAG21_ADDRESS)) {
    sensor21Status = 0x01;
  } else sensor21Status = 0x00;
  if (bag2Sensor.begin(BAG22_ADDRESS)) {
    sensor22Status = 0x01;
  } else sensor22Status = 0x00;

  // Function: readTempC
  // Description: Ping MCP9808 sensor for temperature in C
  // Inputs:
  //   sensor - the sensor to read from
  // /Output: Temperature reading in degrees C
  // Binary float tempSensors::readTempC(const Adafruit_MCP9808 &sensor)
  // Returns:
  //   value: floating point - sensor.readTempC();
  //   return value;}

  static byte tempReadings[16] = {0};
  // Obtain temperature readings from each sensor in C
  if (sensor11Status == 0x00) {
    bag11TempC = readTempC(bag11Sensor);
    tempReadings[0] = bag11TempC.binary[0];
    tempReadings[1] = bag11TempC.binary[1];
  } else {
    tempReadings[0] = BAG11_ADDRESS;
    tempReadings[1] = BAG11_ADDRESS;
    tempReadings[2] = BAG11_ADDRESS;
    tempReadings[3] = BAG11_ADDRESS;
  }

  if (sensor12Status == 0x00) {
    bag12TempC = readTempC(bag12Sensor);
    tempReadings[4] = bag12TempC.binary[0];
    tempReadings[5] = bag12TempC.binary[1];
    tempReadings[7] = bag12TempC.binary[3];
  } else {
    tempReadings[7] = BAG12_ADDRESS;
  }
```
```c
} else {
    tempReadings[7] = BAG_12_ADDRESS;
}

if (sensor21Status == 0x00) {
    bag21TempC = readTempC(bag21Sensor);
    tempReadings[8] = bag21TempC.binary[0];
    tempReadings[9] = bag21TempC.binary[1];
    tempReadings[10] = bag21TempC.binary[2];
} else {
    tempReadings[8] = BAG_21_ADDRESS;
    tempReadings[9] = BAG_21_ADDRESS;
    tempReadings[10] = BAG_21_ADDRESS;
}

if (sensor22Status == 0x00) {
    bag22TempC = readTempC(bag22Sensor);
    tempReadings[12] = bag22TempC.binary[0];
    tempReadings[13] = bag22TempC.binary[1];
    tempReadings[14] = bag22TempC.binary[2];
    tempReadings[15] = bag22TempC.binary[3];
} else {
    tempReadings[12] = BAG_22_ADDRESS;
    tempReadings[13] = BAG_22_ADDRESS;
    tempReadings[14] = BAG_22_ADDRESS;
    tempReadings[15] = BAG_22_ADDRESS;
}

return tempReadings;
```
#ifndef userInput_h
#define userInput_h

#include <Arduino.h>

const byte UP_SWITCH = 4;
const byte DOWN_SWITCH = 9;
const byte SELECT_SWITCH = 2;
const byte BACK_SWITCH = 5;

class userInput {
   //interface
   public:
      userInput(void);
      void initialize(void);
      byte* getStatus(void);

   private:
      byte upSwitch;
      byte downSwitch;
      byte selectSwitch;
      byte backSwitch;
   }

#endif
```cpp
#include <userInput.h>

userInput::userInput(){}

//Function: pinInit
//Description:
//Initializes the following pins as active low inputs:
// -Pressure Switch 1
// -Pressure Switch 2
// -Door Switch
// -Up Switch
// -Down Switch
// -Select Switch
// Sets the pwm frequency to default (31Hz)
// Initializes values to set:
// -Motor On/Off State
// -Fan PWM
// -Heater PWM
// Initializes communication with temperature sensors:
// -Bag 1
// -Bag 2
// -Inlet
// -Outlet
// Inputs: None
// Output: None

void userInput::initialize(void){
    //Configure pins for active high input
    pinMode(UP_SWITCH, INPUT);
    pinMode(DOWN_SWITCH, INPUT);
    pinMode(SELECT_SWITCH, INPUT);
    pinMode(BACK_SWITCH, INPUT);
}

byte* userInput::getStatus(){
    static byte status[4] = {0};
    upSwitch = digitalRead(UP_SWITCH);
    downSwitch = digitalRead(DOWN_SWITCH);
    selectSwitch = digitalRead(SELECT_SWITCH);
    backSwitch = digitalRead(BACK_SWITCH);
    status[0] = upSwitch;
    status[1] = downSwitch;
    status[2] = selectSwitch;
    status[3] = backSwitch;
    return status;
}
```